DEPLOYING DEEP NEURAL NETWORKS IN EDGE WITH DISTRIBUTION

A Dissertation
Presented to
The Academic Faculty

By

Ramyad Hadidi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
Collage of Computing
School of Computer Science

Georgia Institute of Technology

May 2021

DEPLOYING DEEP NEURAL NETWORKS IN EDGE WITH DISTRIBUTION

Thesis committee:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Saibal Mukhopadhyay
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Alexey Tumanov
School of Computer Science
Georgia Institute of Technology

Dr. Michael S. Ryoo
Department of Computer Science
Stony Brook University

Date approved: April 21, 2021

اسرار ازل نه تو دانی و نه من وین حرف معما نه تو خوانی و نه من

هست از پس پرده گفتگوی من و تو چون پرده برافتد نه تو مانی و نه من

The secrets eternal neither you know nor I
And answers to the riddle neither you know nor I
Behind the veil there is much talk about us, why
When the veil falls, neither you remain nor I

Omar Khayyam

To those who did not have the opportunities that life offered me, may this be a testament to what you could have achieved if you had similar opportunities.

ACKNOWLEDGMENTS

I am deeply indebted to my adviser, Dr. Hyesoon Kim, for her unwavering support, unparalleled guidance, and kindheartedness. There are no words to convey my appreciation for her help during my graduate study. I would like to express my deepest gratitude to my committee. The completion of my dissertation would have not been possible without the support of Dr. Michael S. Ryoo. I am grateful to Dr. Tushar Krishna for his encouragement and guidance. I would like to thank Dr. Saibal Mukhopadhyay for his practical suggestions. I appreciate Dr. Alexey Tumanov role in introducing me to systems research.

I also had the great pleasure of working with Dr. Sudhakar Yalamanchili whose advice on my future career path is still guiding me. I am grateful to Dr. Moinuddin K. Qureshi whose wit always enkindles my curiosity. Special thanks to Dr. Jeffery Young for his kind help in presenting our work overseas. Thanks to Dr. Ashutosh Dhekne for his genuine support for my academic applications. I cannot leave Georgia Tech without mentioning Jane Chisholm who helped me tremendously with my writing and presentation skills. I cannot thank enough Dr. H. Venkateswaran for his help during my first year. I thank Dr. Hadi Esmaeilzadeh and Dr. Milos Prvulovic for their assistance in my Ph.D. qualification exam.

I cannot begin to express my thanks to my friends and collaborators Bahar Asgari, Dr. Lifeng Nai, Jiashen Cao, Dr. Hyojong Kim, and Sam Jijina for their unparalleled knowledge and support. Particularly helpful to me during this time were Prasun Gera, John Krzyzaniak, Pranith Kumar, Dr. Jaewoon Sim, Dr. Chad Kersey, Nima Shoghi, Dilan Manatunga, Dr. Joo Hwan Lee, Dr. Sunjae Park, Dr. Ching-Kai Liang, Dr. Jen-Cheng Huang, Jaewon Lee, Yonghae Kim, Blaise Tine, Adriana Amyette, Euna Kim, Dr. He Xiao, Dr. Xinwei Chen, Dr. Karthik Rao, Andrei Bersatti, Dr. Burhan Ahmad Mudassar, and Kartikay Garg.

I am extremely grateful to my family and their unconditioned love and support. I will be ever grateful to my mother, and I am sorry that she has not lived to see me graduate.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

SUMMARY

The widespread applicability of deep neural networks (DNNs) has led edge computing to emerge as a trend to extend our capabilities to several domains such as robotics, autonomous technologies, and Internet-of-things devices. Because of the tight resource constraints of such individual edge devices, computing accurate predictions while providing a fast execution is a key challenge. Moreover, modern DNNs increasingly demand more computation power than their predecessors. As a result, the current approach is to rely on compute resources in the cloud by offloading the inference computations of DNNs. This approach not only does raise privacy concerns but also relies on network infrastructure and data centers that are not scalable and do not guarantee fast execution.

My key insight is that edge devices can break their individual resource constraints by distributing the computation of DNNs on collaborating peer edge devices. In my approach, edge devices cooperate to conduct single-batch inferences in real-time while exploiting several model-parallelism methods. Nonetheless, since communication is costly and current DNN models capture a single-chain of dependency pattern, distributing and parallelizing the computations of current DNNs may not be an effective solution for edge domains. Therefore, to efficiently benefit from computing resources with low communication overhead, I propose new handcrafted edge-tailored models that consist of several independent and narrow DNNs. Additionally, I explore an automated neural architecture search methodology and propose custom DNN architectures with low communication overheads and high parallelization opportunities. Finally, to increase reliability, decrease susceptibility to short disconnectivity or losing a device, I propose a coded distributed computing recovery method that enables distributed DNN models on edge devices to tolerate failures and not lose time-sensitive and real-time information.

CHAPTER 1
INTRODUCTION

The availability of larger datasets, improved algorithms, and increased computing power is rapidly advancing the applications of deep neural networks (DNNs). This advancement has extended the capabilities of deep learning to areas such as computer vision [1], natural language processing [2], neural machine translation [3], and video recognition [4, 5]. DNNs are widely used today in numerous applications, from recommender systems [6] to autonomous vehicles [7, 8]. However, the execution platform of several of DNN applications lacks the resources for the efficient execution of DNNs, such as inexpensive robots [9, 10, 11], unmanned aerial vehicles (UAVs) [12, 13], and Internet-of-things (IoT) devices [14], which are the examples of edge devices. In fact, several of edge devices have access to an abundance of data from their environment and are in desperate need to extract useful information for enhanced handling of complex situations.

1.1 The Key Challenge

While edge devices can tremendously benefit from DNNs, the key challenge is that fast and accurate inferencing requires high compute resources and memory demands [15] that contradicts the limited energy and computational resources of edge devices (*i.e*. resource-constrained devices) [16]. This is because the fast prediction of DNNs (*i.e*., inferencing) is a resource-intensive task [15] that requires energy, large memories, and capable processors. This challenge are even exacerbated in resource-constrained devices.

1.2 Limitation of Current Solutions

The traditional solution to this problem is to offload all the computations to the cloud. Nevertheless, such offloading is not possible in several situations because of privacy concerns [17, 18, 19, 20], limited Internet connectivity, or tight-timing constraints (*e.g.*, home video recordings, drones, and robots surveying a disaster area). Thus, a significant amount of research efforts has been invested to overcome the challenge of locally executing DNNs [21, 22, 23, 24, 25], such as collaborative computation between edge devices and the cloud [26, 27, 28], or customized mobile implementations [29, 30, 31, 32, 33, 34, 35]. Despite all these efforts, scaling current DNNs to edge devices and processing generated data in real time faces challenges due to limited computing power and energy supplies. Additionally newer DNN models encapsulate more parameters and perform more computations for better and more generalized feature understanding than their predecessors. Hence, in order to handle current and future DNN applications that are more resource hungry [36, 37, 38] and extract useful information from raw data in a timely manner, creating an efficient solution is critical.

1.3 The Key Insight

Our approach to deal with the limited resources on edge devices is utilizing distributed computing. Our main insight is to utilize the aggregated computational power of edge devices in a distributed system to perform DNN-based recognition in real time. Such collaboration enables edge devices to take advantage of the collective computing power of the group in an environment to understand the collected raw data. By moving the computations of DNNs to the edge, we achieve the following: (i) reducing the dependence on cloud resources and high-quality network infrastructure for scenarios with limited Internet connectivity such as drones and robots in a disaster area, (ii) improving the privacy of private data since the data is not exposed outside the local network, and (iii) providing an alternative solution to understand raw data locally than the current de facto solution of offloading to the

cloud. Therefore, *by distributing the computation of DNNs, edge devices can collaborate to improve their real-time execution performance.*

1.4 Contributions

To distribute the computations of current DNN models, we explore both data and model parallelism, where data parallelism consists of processing independent data concurrently and model parallelism consists of splitting the computation across multiple robots. In data parallelism, the entire model is duplicated on each device for performing separate inferences. Hence, the system needs several live and concurrent inputs to be efficient without real-time jitter. Simply put, data parallelism only increases throughput and does not reduce the latency per inference. On the other hand, model parallelism divides a model and distributes its computations across several devices for the same inference. In doing so, these methods reduce memory usage and computations, which both are limited in edge devices.

1.4.1 Distributing the Computations

In Chapter 4, we introduce several model-parallelism techniques for DNN models, mainly focusing on computer vision models, convolution neural networks or CNNs. For task assignment, we develop two techniques, offline and online. Our offline profiling-based technique to effectively distribute DNN-based applications on a distributed system while considering memory usage, communication overhead, and real-time data processing performance. Our online technique generates, deploys, and monitors a balanced data-processing pipeline that efficiently processes the computations of DNNs, while requiring significantly less profiling. For experiments, we study prevalent models such as image recognition AlexNet [1], VGG16 [37], ResNet [36], and Xception [39]) and video recognition C3D [40], Ryoo et al. [4]. For experiments, we deploy examples of such systems on an interconnected network of up to 12 Raspberry Pi 3s (RPi) [41]. The following summarizes the contributions of this chapter:

- We introduce several model-parallelism techniques for DNN models, mainly used in computer vision, to reduce the memory footprint per device and divide their computations.

- We propose a system in which collaborative and resource-constrained IoT devices and robots perform the single-batch computation of DNNs in a distributed fashion.

- We develop a profiling-based technique to effectively distribute DNN-based applications on a distributed robot system while considering memory usage, communication overhead, and real-time data processing performance.

- We generate, deploy, and monitor a balanced data-processing pipeline that efficiently processes the computations of DNNs. Our heuristic requires significantly less profiling and exploration than the above profiling-based technique.

- We propose a technique that dynamically adapts to the number of available collaborative robots and IoT devices.

- Our exhaustive experiments performed on real system of up to 12 connected devices deploys prevalent models such as image recognition (AlexNet [1], VGG16 [37], ResNets [36], and Xception [39]), video recognition (C3D [40] and Ryoo et al. [4]), and object detection (YOLO [42]).

1.4.2 Customizing Models for Efficient Distribution

Both data- and model-parallelism cannot reduce communication, memory usage, and computations at the same time. Additionally, although model parallelism could decrease execution latency, it incurs high communication latencies. To address resulted high latency with model parallelism, we explore two custom model designs. First, we propose a low-communication parallelization (LCP) method in which models consist of several almost-independent and narrow branches. In section 5.1, LCP offers close-to-minimum communication overhead with

better distribution and parallelization opportunities while significantly reducing memory footprint and computation compared to data- and model-parallelism methods. Additionally, we propose a new microarchitecture for in-the-edge DNN accelerator. As experiments, we evaluate LCP models based on computer vision DNNs on MNIST [43], CIFAR10/100 [44], Flower102 [45], and ImageNet [46] datasets, and measure their performance real-world implementations on two systems: a system with up to ten Raspberry Pis (RPis), and another with two PYNQ boards. We also evaluate the performance of LCP models on a our own customized hardware (tailored for low latency) implemented on a small edge FPGA and as a 16mW 0.107mm^2 ASIC @7nm chip. LCP models achieve a maximum and average speedups of 56x and 7x, compared to the originals, which could be improved by up to an average speedup of 33x by incorporating common optimizations such as pruning and quantization. The following summarizes the contributions of LCP:

- We propose the first DNN parallelization method to reduce the communication overhead for distributed inference.
- We generate LCP models, with inter-layer parallelism for fast inference at small memory and computation footprints.
- We investigate the impact of hardware/software co-design on inference performance, by tailoring the hardware of TPU [47] for optimizing single-batch inference latency, and implement it on a small FPGA and as a tiny 0.107mm^2 low-power chip consuming only 16mW.
- We conduct real-world experiments on distributed Raspberry Pis, PYNQ boards, and AWS instances.
- We design an edge-tailored TPU-like architecture targeting latency and perform experiments on FPGA. Further, we implement our design on a 16mW 0.107mm^2 ASIC@7nm.

- We generate and evaluate LCP models based on image-recognition DNNs on MNIST [43], CIFAR10/100 [44], Flower102 [45], and ImageNet [46] datasets (total of 53 training results), covering all MLPerf [48] image-recognition models.

Second, while LCP method designs models that are more hand crafted, in section 5.2, we explore a neural architecture search (NAS) methodology to design even better models. The current approaches in decreasing latency only increase parallelism within a layer. This is because architectures typically capture a single-chain dependency pattern that prevents efficient distribution with a higher concurrency (*i.e.*, simultaneous execution of one inference among devices). Such single-chain dependencies are so widespread that even implicitly biases recent NAS studies. We draw attention to an entirely new space of NAS that relaxes the single-chain dependency to provide higher concurrency and distribution opportunities. To quantitatively compare these architectures, we propose a score that encapsulates crucial metrics such as communication, concurrency, and load balancing. Additionally, we propose a new generator and transformation block that consistently deliver superior architectures compared to current state-of-the-art methods. Finally, our results show that these new architectures reduce the inference latency and deserve more attention. Our results on MNIST and Flower102 dataset show that these novel models achieves 6–7x in execution performance compared to previous work. The following summarizes the contributions of section 5.2:

- Addressing Single-Chain Data Dependencies: Our concurrent architectures created by network generators (specially the new distance-based generator) break current biased designs by delivering high concurrency.

- Proposing Representative Concurrency Score: Our problem formulation based on hypergraph theory encapsulates critical metrics to quantitatively compare all architectures for efficient distribution and concurrency.

- Conducting Comprehensive Experiments: Our results show that our new models

achieves 6–7x in execution performance compared to previous work.

1.4.3 Increasing Reliability

These promising methods harvest the aggregated computing power of the edge devices in an environment. However, since such a distributed system strongly relies on each device, unstable latencies, and intermittent failures, the common characteristics of edge devices and wireless networks, cause high recovery overheads. To reduce this overhead, in the last part of this thesis, Chapter 6, we propose a novel robustness method with a close-to-zero recovery latency for DNN computations. Our solution never loses a request or spends time recovering from a failure. To do so, first, we analyze the underlying matrix-matrix computations affected by distribution. Then, we introduce a new coded distributed computing (CDC) method that has a constant cost with the increasing number of devices, unlike the linear cost of modular redundancies. Moreover, our method is applied in the library level, without requiring extensive changes to the program, while still ensuring a balanced work assignment during distribution. To illustrate our method, we perform experiments with distributed systems comprising up to 12 Raspberry Pis. The following summarizes the contributions of our CDC-based method:

- We thoroughly analyze how general methods of distributing the computation of DNNs affect the underlying matrix-matrix computations.
- We propose a novel fault recovery method based on CDC that has close-to-zero recovery latency, does not disturb the balanced work assignment in distribution, requires minimal changes to the user's program, and has a constant cost with the increasing number of devices.
- We demonstrate our method on distributed systems of up to 12 Raspberry Pis and report our experimental results.

CHAPTER 2
BACKGROUND & MOTIVATION

This chapter first provides background on common layers used in deep neural network (DNN). Our focus is mostly on a prevalent type of DNNs convolution neural networks (CNNs) that are used in computer vision. Then, we introduce the model architecture of some well-known models that are used in the rest of this document. Finally, we discuss about why in-the-edge inferencing is important and motivate why distributing the computation of one inference computation is necessary.

2.1 DNN Layers

We provide a background on the layers that are currently being used in prevalent DNNs, which are (i) multilayer perception (MLP), (ii) recurrent neural network (RNN), (iii) long short-term memory (LSTM), and (iv) convolution neural network (CNN). We introduce the computations of the most compute- and data-intensive layers in these DNNs [49, 50], with of focus on CNNs. This introduction is crucial for understanding model-parallelism methods. While providing an overview on the type of computation in these layers, we describe how these computations are done in underlying matrix-matrix multiplication libraries (*i.e.*, GEMM) [51, 52, 53]. GEMM kernels are the key computation done in almost all the DNNs. Therefore, understanding how these computations are done in common libraries is important in designing efficient model-parallelism methods.

2.1.1 Fully-Connected Layer

A common layer found in almost all DNNs, from MLPs to CNNs and RNNs, is the fully-connected (`fc`) layer. A fully-connected layer has several outputs, each of which represents a neuron. An output *fires* when the weighted sum of its input values is greater than a

threshold. After a neuron fires, the amount of the output or *activation* is determined by an activation function, such as sigmoid ($1/1+e^{-x}$) or ReLU ($max(0, x)$) functions. So, we can write a single activation as

$$a = \sigma\Big(\sum_i x_i w_i + b\Big), \tag{2.1}$$

in which a is the activation, σ is the activation function, inputs are denoted as x_i, weights per input as w_i, and bias (*i.e.*, negative of threshold) as b. In a model, there are several layers; therefore, we can write the computation of a single layer based on the activation of its previous layer as:

$$a^l_j = \sigma\Big(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\Big), \tag{2.2}$$

in which a^l_j is the j^{th} activation of the l^{th} layer, w^l_{jk} is the weight from k^{th} input in the $(l-1)^{\text{th}}$ layer to the j^{th} output in the l^{th} layer, and b^l_j is the bias of the j^{th} output in the l^{th} layer. Note that the notation of w^l_{jk} is from k to j. Figure 2.1 shows an example for w^3_{24}, w^2_{31}, and a^3_2. This notation is useful when we write the computations in the matrix format. In fact, we can write all the multiplications of the l^{th} layer in Equation 2.2 as the below matrix operation:

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mk} \end{bmatrix}_{m\times k} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix}_{k\times 1} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}_{m\times 1}, \tag{2.3}$$

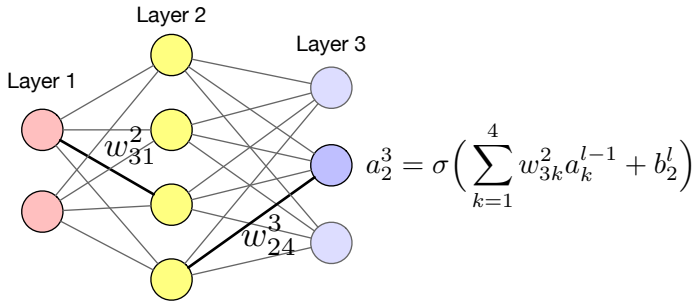


Figure 2.1: **3-Layer MLP Example** – The MLP consists of fully-connected layers, and one example of its output computation. Note that the notation for weights, w^l_{jk}, is from k to j. This notation helps us in writing a simpler matrix notation.

in which m is the number of output elements, k is the number of input elements, and a' represents previous layer activations (activation function and bias is removed for simplicity). As we see, the weight notation helped us in writing a weight matrix or $\mathbf{W}$, in which its i^{th} row represents weights needed to compute i^{th} output. Thus, we can write the computations in the l^{th} layer as

$$\mathbf{a^l} = \sigma(\mathbf{W^l a^{l-1}} + \mathbf{b^l}). \tag{2.4}$$

By stacking a few layers of these layers, an MLP could learn complex functions. This learning is done by adjusting $\mathbf{W}$ and $\mathbf{b}$ during training. Furthermore, note that the computations of Equation 2.4, in its current format, can easily utilize GEMM libraries [51, 52, 53]. Thus, no transformation is needed to execute fully-connected layers.

2.1.2 Convolution Layer

The main layer in CNN models that process any kind of visual data is the convolution layer. In fact, all the layers except the last ones are convolution layers `(conv)`. A convolution layer applies the same set of weights (*i.e.*, filters) to subsets or patches of input. In a scenario in which the input is two dimensional (2D), such as an image, each filter is swept across the image, shown in Figure 2.2. For each output element, the elements of the corresponding input patch and the filter weights perform similar operations 0as a single output neuron in a fully-connected layer. Figure 2.2 depicts the convolution of an input with size H_ixW_ixC_i with K square filters of size FxFxC_i. Each filter creates a channel, or depth (*i.e.*, z-axis), of the output. Thus, the depth of the output, C_o, is equal to the number of filters, K, $C_o = K$. The height and width of the output are determined by how a filter is swept across the input by parameters such as stride (s), filter size (f), and padding (p). In short, the output size in any dimension is derived from $\lfloor {}^{i-f+2p}/_{s} \rfloor$, in which i is the input size in that dimension. For the sake of simplicity, in this paper, we assume the same padding condition, so that the size of the output height and width is the same as the input. In other cases, one can simply replace the output dimensions with the above formula. Essentially, a convolution layer is a

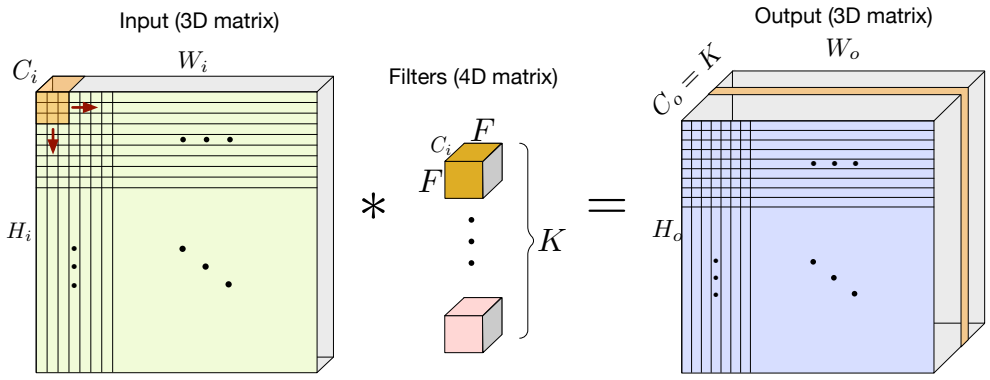
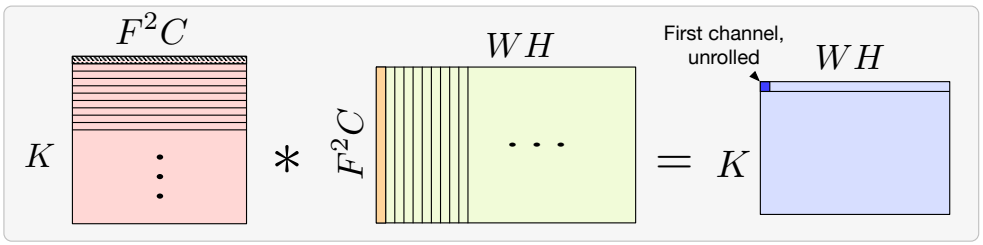


Figure 2.2: **Visualization of Convolution Layer Operation** – Input size of H_ixW_ixC_i, K filers of size FxFxC_i, and output size of H_oxW_oxC_o. Each filter creates a depth channel in the output; thus $C_o = K$.

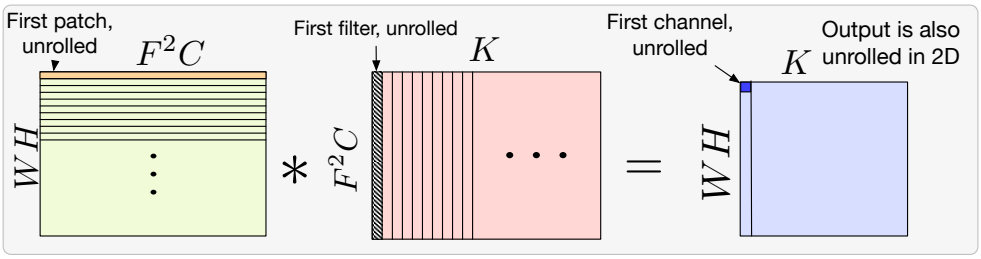
special case of fully-connected layers, but since visual features are invariant to their location in a picture (*e.g.*, a cat is always a cat regardless of its location in the picture), instead of separate and independent weights for each input, inputs share the same set of weights.

As discussed, the computations of a convolution operation consist of several overlapped matrix-matrix multiplications of input patches and filter weights. To apply our idea of coded distributed computation for robustness in a straightforward way, it is beneficial to transform these computations as a single matrix-matrix multiplication. In fact, almost all machine learning frameworks perform this transformation to harness the power of extensively optimized GEMM libraries and simplify the convolution operation [51, 52, 53]. The essence of the transformation is to unroll the input patches (a 3D matrix) and filters (a 4D matrix) in 2D in a way that a single matrix-matrix multiplication produces the unrolled version of the output in 2D. Thus, one can apply GEMM instead of convolution [51, 52]. To do this, we need to convert the weight matrix of the filters, which is a 4D matrix of size FxFxCxK, to a 2D matrix of size KxF^2C. Similarly, the 3D input matrix of size WxHxC is converted to a 2D matrix of size F^2CxWH by unrolling each patch and repeating the overlapping elements (if necessary). Figure 2.3a illustrates the unrolling operation we discussed, which transforms convolution as a single matrix-matrix multiplication as below:

$$\mathbf{O}_{K\text{x}WH} = \mathbf{W}_{K\text{x}F^2C} \times \mathbf{I}_{F^2C\text{x}WH}. \tag{2.5}$$



(a) $\mathrm{W}_{k \times F^2C} \times \mathrm{I}_{F^2C \times WH} = O_{K \times WH}$



(b) $\mathrm{I}_{WH \times F^2C} \times \mathrm{W}_{F^2C \times K} = O_{WH \times K}$

Figure 2.3: **Convolution Layer Operation in GEMM** – Underlying transformation of a convolution operation to use GEMM libraries. Each method unrolls input patches and filters to create output, which is also produced unrolled.

Figure 2.3b illustrates another possible way to transform the convolution operation. As seen, after such transformations, the type of computations (*i.e.*, Equation 2.5) for a convolution layer is equivalent to that for a fully-connected layer. Since machine learning frameworks already perform these transformations, for convolution layers, we build our robustness technique on top of these transformations.

2.1.3 Other Layers

To introduce non-linearity, an activation layer (φ), such as ReLU, is applied on the output to create the input to the next layer, or $h_j = \varphi(a_j)$. This allows a model to learn complex functions. Although, there are several activation functions, modern DNNs in computer vision usually use sigmoid ($1/1+e^{-x}$) or ReLU ($max(0, x)$) functions.

In addition, to activation functions, often a pooling layer downsamples the data and reduces the dimensions, such as a max pooling (`maxpool`) or average pooling (`avgpool`) layers. The main purpose of such layers are to reduce the computational complexity of the next layer by reducing the dimensions of the input. Moreover, these layers sometimes

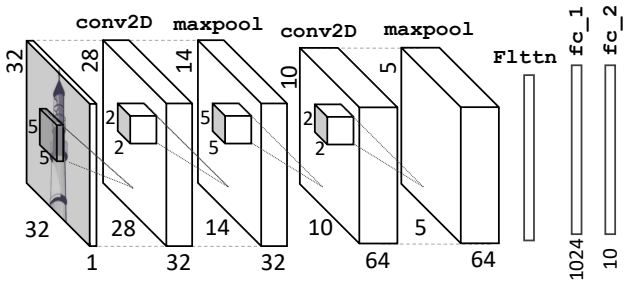


Figure 2.4: **LeNet Model** – LeNet-5 model [54] architecture for digit recognition.

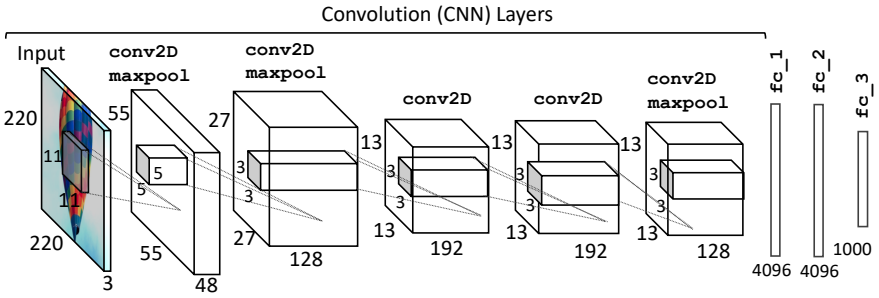


Figure 2.5: **AlexNet Model** – AlexNet image-recognition model [1].

aid in the learning process by reducing the size of the feature space. Compared to `fc` and `conv` layers, pooling layers are much less compute intensive, so we group them with their corresponding parent layer in our studies.

2.2 DNN Models

This section introduces the architecture of the common DNN models. All the introduced models are considered as CNNs. We choose well-known models that are usually used in other studies.

LeNet: LeNet-5 [54] is a straightforward CNN with a goal of recognizing handwritten digits. It consists of two sets of convolution layers, each of which is followed by a pooling layer, then a flattening layer, and then two fully-connected layers. The input is a picture of a digit with the size of 32x32x1. The activation function used in the model is *tanh*, and for final probability calculation uses `softmax`. Figure 2.4 illustrates the LeNet architecture and its parameters.

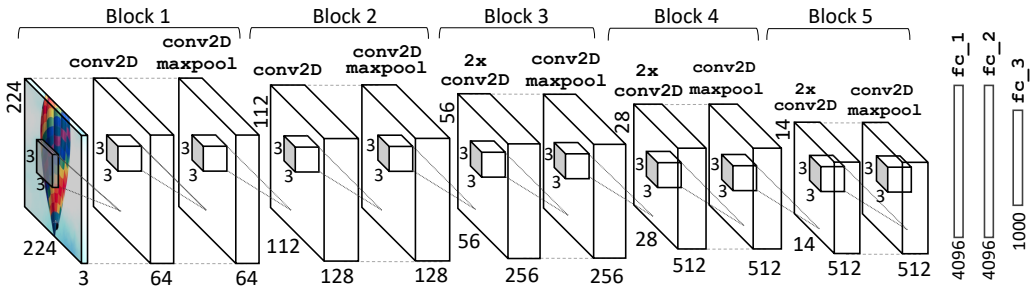


Figure 2.6: **VGG16 Model** – VGG16 image-recognition model [37].

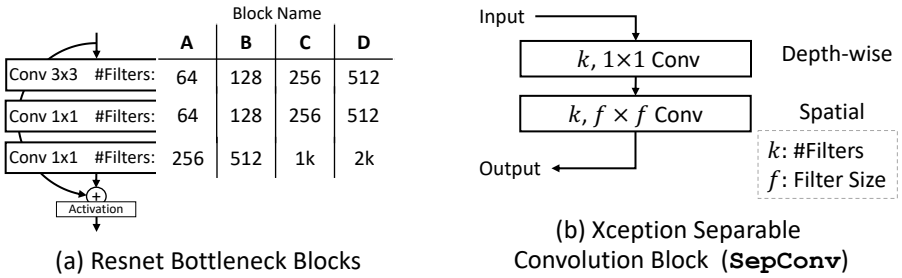


Figure 2.7: **Residual Building Blocks** – Building blocks of Resnet50 and Xception [36, 39].

AlexNet: In the 2012 ImageNet large-scale visual recognition challenge (ILSVRC), AlexNet-[1] significantly outperformed all the prior competitors. Figure 2.5 illustrates the model of a single-stream AlexNet, which consists of five convolution layers and three fully-connected layers. The model that we use has around 40M parameters (single-stream AlexNet).

VGG16: Figure 2.6 depicts the VGG16 model [37], which has total of 16 layers: 13 convolution and three fully-connected layers. As seen, VGG16 has a structured model; deeper convolution layers have more filters and smaller spatial dimensions.

ResNet: The residual neural network (ResNet) [36] introduced "skip-connection" for training deeper networks in 2016. In this paper, we used ResNet50 with 50 layers (Figure 2.8). Additionally, Figure 2.7a illustrates the basic blocks for ResNet that are used in Figure 2.8. This model is residual in the sense that shortcut connections skip some blocks, which makes training easier for deep models.

Xception: The Xception [39] model is based on Inception V3 [38]. The Xception module independently processes the correlations in cross-channel and spatial features. Therefore,

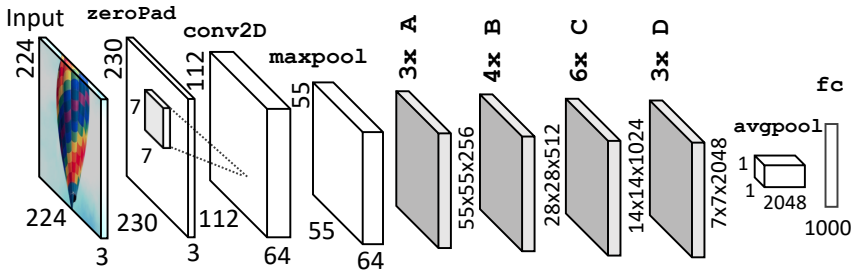


Figure 2.8: **ResNet50 Model** –ResNet50 image-recognition model [36].

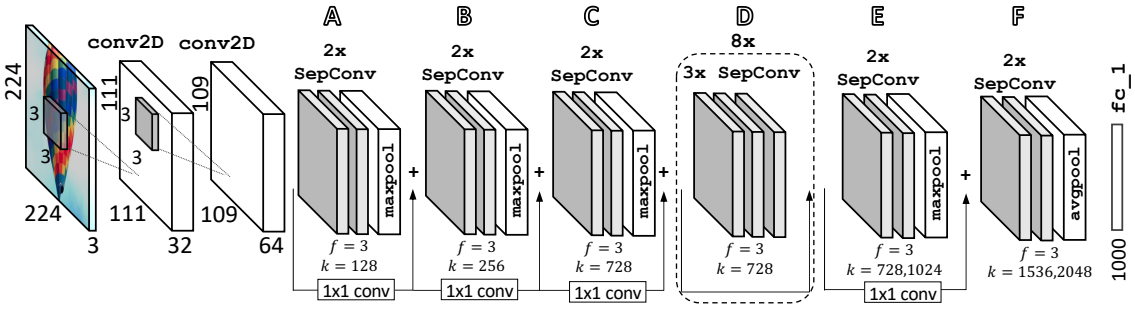


Figure 2.9: **Xception Model** – Xception image-recognition model [39].

Xception introduces a special convolution unit, shown in Figure 2.7b, the separable convolution unit that decouples the mapping of cross-channel and spatial features. Separable convolution first performs cross-channel (*i.e.*, depth-wise) convolution over input channels, and then performs an independent spatial convolution on each of the outputs. Figure 2.9 shows the Xception model with 34 separable convolution layers.

C3D: The C3D [40] model is designed to process videos and has been used in action recognition and scene classification tasks. To learn spatio-temporal features, the C3D model uses 3D convolutions, which produce an output volume instead of a 2D output per filter. Compared to a conventional convolution layer, an additional sweep along the z-axis creates a volume in the output. Figure 2.10 shows the C3D model, which consists of eight 3D convolution layers.

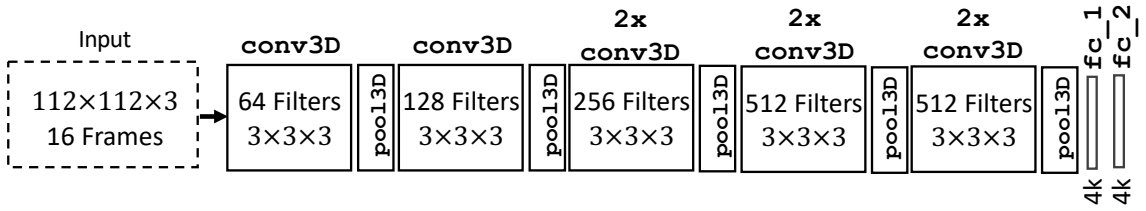


Figure 2.10: **C3D Model** – C3D action-recognition model [40].

2.3 In-The-Edge Inferencing

Deep and convolution neural networks (DNNs/CNNs) have provided impressive performance improvements and solutions to traditionally hard problems [55, 56, 57]. Because of such advancements several domains such as robotics [9, 10, 11], unmanned aerial vehicles (UAVs) [12, 13], and Internet-of-things (IoT) devices [58, 14, 59] are undergoing rapid changes. In such domains, *inferencing in-the-edge* is rapidly gaining ground, due to ubiquitous wireless networks, the availability of embedded processors, and numerous applications of deep learning. For instance, the capabilities of DNNs can highly aid drones that grab a huge amount of local data or Internet-of-things (IoT) devices such as cameras in a city that generate large streams of data.

2.3.1 Growing DNNs & Resource Limitation

Although in-the-edge inferencing is beneficial from many aspects, the key challenge is that fast and accurate inferencing requires high compute resources and memory demands [15] that contradicts the limited energy and computational resources of edge devices (*i.e.* resource-constrained devices) [16]. Such demands are not expected to slow down as modern models [60] encapsulate more parameters for a better generalization. Additionally, privacy concerns [17, 18, 19], unreliable connection to the cloud, tight real-time requirements, and personalization are pushing inferencing to the edge.

DNN models consist of several layers, each of which performs specific computations. The computations are based on custom weights that are learned during the training phase

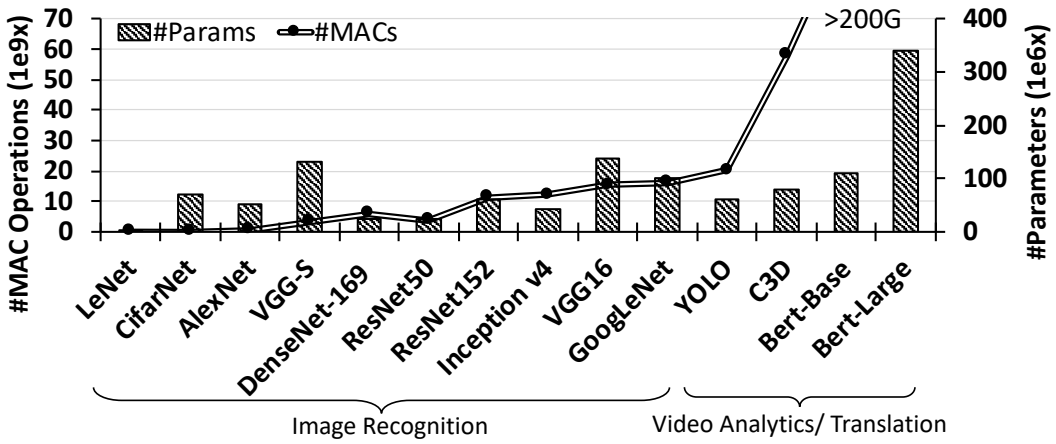


Figure 2.11: **DNNs Computational Trend** – Number of MAC operations/inference and parameters of modern DNN models.

with backpropagation. In the inference phase (*i.e.*, prediction), feed-forward computations are performed on batched inputs and learned parameters stay constant. For edge computation, the most compute- and data-intensive layers [49, 50] are fully-connected and convolution layers.[1]In fact, DNNs are inherently compute-intensive; Figure 2.11 shows the amount of multiply-accumulate operations and parameter size in several DNN models. The left bars illustrate basic models such as LeNet [54] and CifarNet [44]. On the right side, we illustrate the YOLO [42] and C3D [40] models that are used for videos. The newest translation model, Bert [60], significantly surpasses all the previous models in both parameter size and computations. As shown, newer models encapsulate more parameters and perform more computations for better and more generalized feature understanding than their predecessors. In short, this trend of modern models will inevitably surpass the capabilities of any resource-constrained device.

2.3.2 Single Device Pareto Frontier

The capabilities of resource-constrained platforms are limited. To gain a better understanding, Figure 2.12 depicts latency per image for state-of-the-art image recognition models for ILSVRC 2012 challenge [46] on RPi [61]. All implementations heavily utilize modern machine learning optimizations such as pruning [15], quantization, low-precision inference [62,

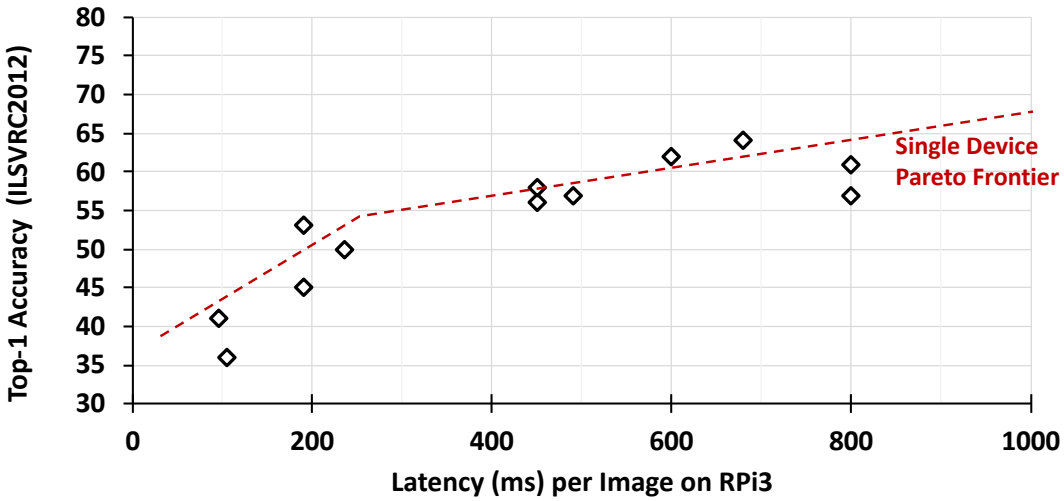


Figure 2.12: **Latency-Accuracy Single-Device Pareto Frontier** – Single device: Latency per image on RPi3 for state-of-the-art ILSVRC models with the optimized platform-specific compilation ELL [29] tool [61].

63, 64], and handcrafted models [35]. Additionally, the models are highly optimized for ARMv8 architectures using the ELL compilation tool [29]. However, achieving higher execution performance is impossible on a single device due to the Pareto frontier. As seen, the latency for high-accuracy models is longer than 400ms, and generally, latencies are longer than 100ms. In addition, the data shown in the figure is only for image-recognition models and DNNs in other domains are already surpassing these models in size and complexity. Fitting such an exponentially increasing computation on a single device, especially for edge devices, is a limiting factor for executing DNNs in the edge. In other words, even after applying all optimization techniques for DNNs in embedded systems, the single device Pareto frontier is still limiting the widespread applicability of DNNs in several in-the-edge applications.

2.3.3 Distribution Methods & Their Limitations

(1) Data parallelism parallelizes the computations of independent inputs. Among distribution methods, data parallelism [1, 65] keeps computation and memory footprints per device similar to the original DNN. Data parallelism does not apply to the edge because: (i) serves

several independent requests, that are limited in an edge environment; (ii) does not reduce latency, important in several real-time applications in the edge; and (iii) does not reduce computation and memory footprints per device.

(2) As we discuss in Chapter 4, model-parallelism methods divide the computations of a DNN model for a request. These methods first divide the computations based on layers in a model and then internally divide the computations within each layer, by keeping dependencies intact. Depending on the type of the layer, the dividing could take several forms. In Figure 2.13, we provide a simple example for distributing a fully-connected (`fc`) layer. The computation of an `fc` layer is as $\sum_i x_i w_i + b$, in which w, x, and b are weights, input, and biases, respectively. We can write this computation as matrix-matrix multiplication, or $\mathbf{Wx} + \mathbf{b}$. There are two extremes of model parallelism, input and output splitting (see section 4.2). In output splitting, producing each set of outputs is divided among the devices. In input splitting, the input is split and each device computes all parts of the output that are dependent on their received input. As shown in Figure 2.13, each technique has specific communication overhead. Output splitting requires the transmission of the input to all nodes. Input splitting requires the transmission of partial sums to a final node for summation. New model-parallelism methods can also be crafted by mixing these two extremes, but they similarly suffer from the same discussed overheads.

Several model-parallelism techniques also exist for convolution layers as we discuss in section 4.3, but they have similar characteristics to the example provided for `fc` layer.

In summary, since model-parallelism techniques do not change the internal network connections of a model, after distribution, we need to keep the dependency chains intact. Hence, although model parallelism reduces the compute and memory footprint per device, the communication overhead resulting from the tightly interconnected layers and inter-layer dependencies stays the same as the original model.

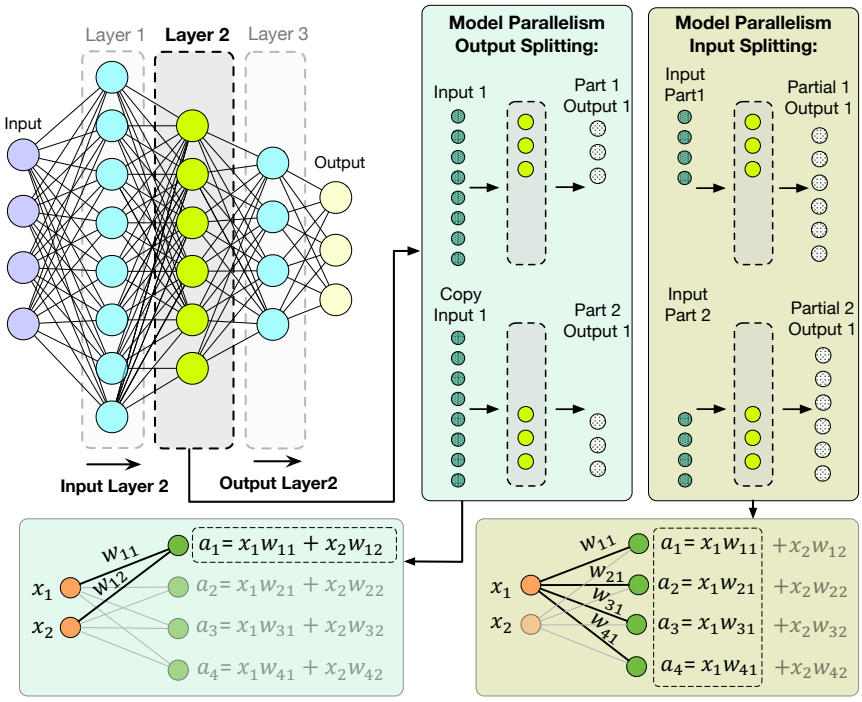


Figure 2.13: **Basics of Model Parallelism** – Distributing a fully-connected layer with input- and output splitting. Note the communication overheads.

2.3.4 Communication Challenges

Dependability current distribution methods on the high amount of communications induce the straggler problem, in which a system is lagged by its slowest node. Specifically, since edge devices usually use a wireless or mobile network, the latency deviations are high. Figure 2.14 depicts the histogram of prediction latencies on a distributed IoT system consisting of six RPis executing AlexNet [66] with model parallelism. The computing time is bounded to 500ms, but the average delay is ≈2x longer (and ≈4x for tail latency). To gain perspective, Figure 2.15a and b depict the VGG-S model and its distributed version with model parallelism, respectively. The VGG-S model has a similar parameter size and compute density as AlexNet [1] (Figure 2.11) and it is designed for the Flower102 [45] dataset. As seen, dependencies enforce a strongly interconnected network among the divided parts. Although several techniques such as compression could alleviate the cost of communication, they do not reduce the number of connections.

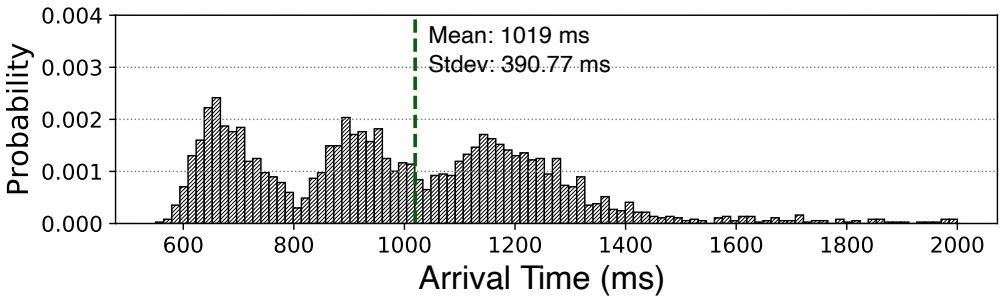


Figure 2.14: **Straggler Problem** – Histogram of prediction latencies on a six Raspberry Pi system executing AlexNet with model parallelism.

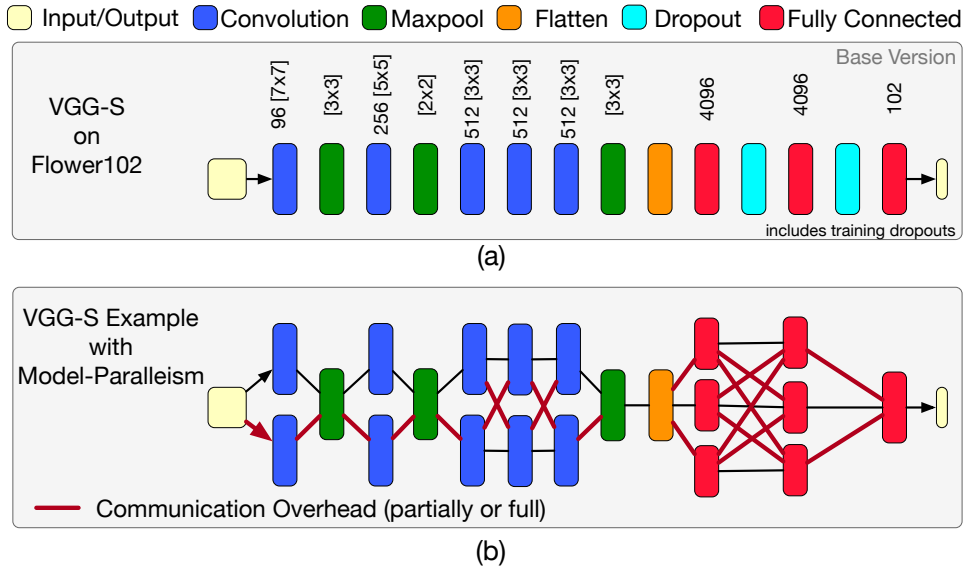


Figure 2.15: **Model Parallelism View for The Entire Model** – (a) VGG-S, (b) and its distributed version.

Relying extensively on communication in model-parallelism techniques also imposes another challenge: finding an optimal distribution. This is because each distribution depends heavily on communication and network traffic that changes over time. Hence, a single distribution is not always the best answer and actively profiling the distribution in real-time is necessary. In particular, finding an optimal distribution is an NP-hard problem. Since our distribution reduces the dependability to communication, it reduces the complexity of the search space. In fact, our models only need to communicate for the input and before the final classification layer activations. Additionally, ensuring reliability in our models is

easier than model-parallelism versions due to the same reasons discussed. In summary, with parallel execution on multiple devices, ideally, we could pass the frontier. But, as will show in our studies, current distribution methods are limited by the communication overhead and the inherent inter-layer data dependency in models.

CHAPTER 3
PRIOR WORK

Edge computing with advantages such as security and local processing has introduced a new paradigm for processing deep neural networks (DNNs) in the edge. To enable the fast and efficient inference of DNNs previous work has proposed several techniques. Although some of the techniques are proposed for DNNs in general, they are useful for increasing the performance of their execution. After discussing such techniques, we introduce specific edge-oriented DNN frameworks and hardware accelerators. Finally, we overview current state-of-the-art neural architecture search studies.

3.1 Computation and Parameter Reduction

Since inferencing is a compute- and memory-intensive task, a group of studies has proposed techniques that trade accuracy (sometimes) with performance to create small models. Smaller models have Reduced computation and size of parameters to increase inference speed. Such techniques are applied after a model architecture is fixed. One common approach is to remove the weak connections with weight pruning [67, 15, 68, 69, 70, 71], in which the close-to-zero weights are pruned away. It is also been shown that moderate pruning with iterative retraining enables superior accuracy [15]. Quantization and low-precision inference [72, 62, 63, 73, 64] change the representation of numbers in hardware for simpler and faster calculations. For inference, it has been shown that instead of FP32, we can use INT8 without any accuracy loss with significant gains in energy savings and speed. [15] Several methods also have been proposed for binarizing the weights [74, 75, 30], which hurts the accuracy of a model. Several of aforementioned techniques that do not change the accuracy are considered a common practice in deploying DNN models for production. However, they require several additional steps that enforce retraining of the model.

3.2 Distribution and Parallelization

With the prevalence of large DNN models, distributing a single model has gained the attention of researchers. Large models need more memory, and when the memory requirement of a DNN model is larger than the system's memory, the performance of a model suffers noticeably. More important, when executing DNNs on edge devices, compared with HPC machines, two important criteria change: (i) we cannot batch several requests and make use of data parallelism and (ii) we do not have access to machines with high memory capacities.

3.2.1 Techniques without Changing Model Architecture

Since training performance is usually a bottleneck, initial studies have tried to distribute the training of a model [1, 65]. These studies either exploit data or model parallelism Data parallelism only increases the throughput of the system and does not affect the latency. Model parallelism divides the work of a single inference among workers. However, model parallelism keeps the connections of a model intact. Thus, applying model parallelism on intra-layer computations results in a huge communication overhead for sharing the partial results after each layer due to existing single-chain dependency. Both studies investigate such distribution and partitioning specific for training and not inference while only focusing on high-performance hardware.

As more focused study on inferencing is Neurosurgeon [26], which dynamically partitions a DNN model between a *single* edge device (Tegra TK1, $200) and the cloud for higher performance and better energy consumption. A similar study of partitioning the computations between mobile and cloud is also done in [27] using the Galaxy S3. Another work, MoDNN [76], creates a local distributed mobile computing system and accelerates DNN computations. MoDNN uses only mobile platforms (LG Nexus 5) and partitions a DNN using input partitioning within each layer, especially by relying on sparsity in the matrix multiplications. However, MoDNN does not consider real-time performance because its

most optimized system with four Nexus-5 devices has a latency of six seconds. DDNN [28] also aims to distribute the computation in local devices. However, in its mechanism, in addition to retraining the model, each sensor device performs the first few layers in the network and the rest of the computation is offloaded to the cloud system. Therefore, similar to [26, 27] is dependent on the cloud.

3.2.2 Techniques with Changing Model Architecture

Several proposals have developed mobile-specific models [31, 35, 77, 78, 79]. The common approach is to handcraft more efficient operations or models. The objective is to reduce the number of parameters [35], create efficient operation to minimize computation density [31], or use resource-efficient connections [79]. Moreover, several of the models tradeoff the state-of-the-art accuracy with efficiency [79]. SplitNet [80] focuses on improving the parallelization opportunity within a model by explicitly enforcing dataset semantics in the parallelizing of *only* the final layers. Each model needs to be handcrafted individually for each dataset by examining the semantics in the dataset. Additionally, SplitNet only creates a tree-based model, which introduces increasingly more merging/synchronization points with higher depth.

3.3 Edge-Specific Frameworks

The importance of in-the-edge inference of DNNs has encouraged researchers and industry to propose several specialized frameworks. Some of these frameworks only work with a specific hardware. For instance, ELL library [29] by Microsoft is optimized for Raspberry Pis. As another example, Movidius Neural Compute SDK (NCSDK) [81] that is developed for Movidius Neural Compute Stick [82] from Intel. On the other hand, some frameworks have considered generality in their platforms such as Tensorflow Lite [34], TensorRT [83], and TVM [84]. Since generality contradicts specialization on each device, it is usual that with increased usability and a wide range of supported platform the gained performance

speedup becomes smaller.

3.4 Edge-Targeted Accelerators

A state-of-the-art systolic-based DNN accelerator is TPUv2 [85], which provides a peak of 180 TFLOPS by employing four dual-core chips, each connected to an 8GB HBM package at 300 GB/s. Many other recent deep-learning accelerators utilize systolic arrays concepts [25, 86, 87, 88, 89, 47, 85, 90], which increase the performance of inference by utilizing sparsity, reducing memory accesses by exploring access patterns, or employing weight-stationary architectures. Several studies also target FPGA/ASIC implementations for DNNs [91, 92, 93, 94, 95, 96].

Finally, several recent academic efforts have proposed custom accelerators [87, 25, 86, 87, 88, 89, 97, 90, 47, 85, 71, 98], which improve inference by utilizing sparsity, reducing memory accesses, or employing efficient dataflows. In addition, many of the custom designs have targeted FPGA/ASIC implementations for inference acceleration [91, 92, 93, 94, 95, 96].

3.5 Neural Architecture Search

Neural architecture search is a method to find optimized and efficient models. Recently, with the growing interests in automating the search space for models and moving away from handcrafted models, several studies [99, 100, 101, 102, 103, 104, 105] have proposed new optimization or searching methods. The goal is to find an efficient model that satisfy some criteria (*e.g.*, minimizing time on mobile CPUs [104]). MnasNet [104], a platform-aware neural architecture search for mobile devices tries to balance a the accuracy of a model and its execution time on mobile devices. They use a hierarchical search space to find models similar to MobileNets [79] that are faster and have lower overheads. Other studies [99, 100] usually utilize an LSTM controller for generating the model architecture. However, as pointed out in [105], the search space in these studies are determined by several implicit

assumptions in network generators and sometimes explicit staging (*i.e*., downsampling spatially while upsampling channels). Xie *et al*. [105] tried to remove all the implicit wiring biases from the network generator by using classical random graph generator algorithms. In their randomly wired paper [105], they illustrate that models that have similar computational complexity (*i.e*. FLOPs) and parameter size achieve similar or better accuracies to well-know handcrafted models. However, they introduced a scaling/staging bias in the final model to deal with a large amount of computation. Such stagings create a merging point after a stage where all the features are collected and downsampled before the next stage. Hence, the generated model carries a single-chain of dependency which limits the parallelization and distribution.

3.6 Coded Distributed Computing

The authors of coded distributed computing [106] studies introduced coding for MapReduce-type workloads for large-scale computing. By coding, which increases the computation load of mapping functions, the amount of communication can be reduced in the reduction phase. The authors theoretically study the limits and tradeoffs of such distribution and illustrate an inverse relationship between the amount of computation and communication. Usually, coding in CDC is applied over bit-level representation of numbers. Instead of coding over floats/bits, our work applies coding to the application level by introducing new weights. Furthermore, in contrast, to reduce communication overhead in other studies, our goal is to increase robustness and tolerating unstable latencies.

CDC helps to mitigate the straggler problem in computing clusters [107, 108], besides other methods such as straggler detection algorithms [109, 110] and replication-based approaches [111, 112]. Several works also utilize CDC to mitigate the straggler problem in distributed storage systems [113]. Distributed learning algorithms have also used CDC opportunities [114]. Since these algorithms use data parallelism for learning, CDC facilitates the mapping phase in learning algorithms with data shuffling. Particularly, Lee et al. [114]

focused on two basic blocks of learning algorithms, matrix multiplication and data shuffling. None of the above works has studied CDC in the context of robustness. In contrast with our work, distributed learning studies [114] examine large-scale learning algorithms, which employ data parallelism, whereas our work focuses on IoT-based inferencing, which utilizes model parallelism.

CHAPTER 4
DISTRIBUTING THE COMPUTATIONS

In this chapter, we present model-parallelism methods for distributing and parallelizing the computations of single-batch inferences in fully-connected and convolution layers. Then, we discuss how we use these model-parallelism methods to distribute the work among several devices. Next, we explain how these systems are expandable to video streams to be used in video action recognition. Finally, we present our experimental results on real systems consisting of Raspberry Pis [115, 116].

4.1 Data & Model Parallelism

For parallelizing computation of a layer, we examine two general directions: data and model parallelism, as shown in Figure 4.1. Data parallelism for DNN models was introduced in early days of recent deep learning explosion for fully-connected layers [1, 65]. The main use of data parallelism is usually for training to increase the throughput. In model parallelism, which is applicable to the computations required for a single input, the inference computations are distributed and parallelized over multiple devices. Data parallelism is realized because of batch processing, grouping several requests together.

In data parallelism, the presence of many requests at the same time enables us to increase the number of inferences per second by independently serving each inference separately on each device. Applying only data parallelism would not always work for edge devices. This is because data parallelism duplicates the same amount of computations on another device. Since computations are the same but on a different input data, the memory and computation footprints are not reduced. In detail, data-parallelism alone cannot ensure high performance in IoT devices because: (i) for large layers, just the duplication of devices does not provide a performance benefit because the entire data is not loaded to the memory. Thus, a device

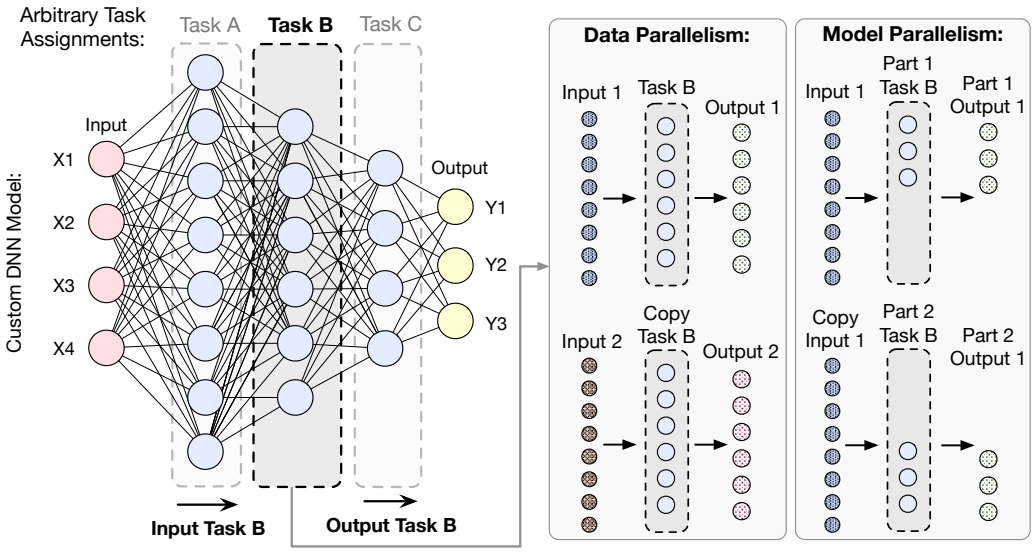


Figure 4.1: **Model and Data Parallelism Example** – A simple example for task B on two devices.

still pays a high cost for accessing the off-chip storage (i.e., swap); (ii) data parallelism needs a stream of input data, whereas in several scenarios, the frequency of input data is low; (iii) to create a balanced and efficient data-processing pipeline in a distributed system, a balanced work assignment is required. However, data parallelism is not flexible in adjusting the amount of computation per device.

Model parallelism, on the other hand, exploits intra-layer independence of computations in DNNs to create fine-grained divisions of work. Thus, it solves the mentioned shortcomings of data parallelism. However, compared to data parallelism, employing such deeper-level parallelism needs knowledge of how each layer does its computations and how parallelism affects data communication, computations, and aggregation. We address this knowledge gap without changing the lower-level implementations or distributing only one model. Our goal is to provide general methods that are applicable on common deep learning frameworks without changing any lower-level implementation.

4.2 Model Parallelism for Fully-Connected Layers

In a fully-connected layer, since the computations of each activation (a_j) are independent of other activations, we can parallelize its computations. We describe two methods specific to fully-connected layers: Output and input splitting, shown in Figure 4.2a and b, respectively. In output splitting, we parallelize the computation of each activation while transmitting all input data to each device. Figure 4.2a highlights a device and its computations to derive its activation. Each device holds the weights corresponding to its activations. Later, when the computations of each device are done, we merge the results by concatenating values in the correct order. We can apply an activation function either on each device or after the merging.

In input splitting, a device computes a partial part of *all* the activations. Figure 4.2b illustrates an example in which a device computes half of the required multiplications for all the activations. In this method, a part of the input is transmitted to each device. Each device holds the weights corresponding to its input split. Later, when the computation of each device is finished, a merge operation adds all of the partial sums. Contrary to the output-splitting method, we cannot apply an activation function before the merge. The mentioned methods may also be mixed, which creates a spectrum of methods; however, we focus on extreme cases of this spectrum.

A more detailed summary of these methods is presented in Table 4.1. These methods trade communication with the memory footprint. This is because each device holds part of

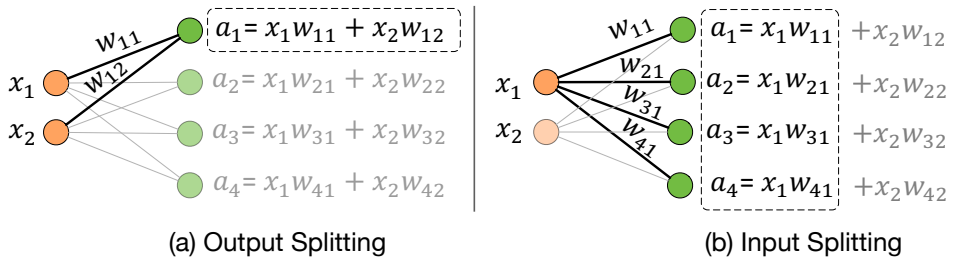


Figure 4.2: **Illustration of Model Parallelism for a Fully-Connected Layer** – Model parallelism methods for a simple fully-connected layer.

the weights but needs to transmit more variables. A more detailed examination is shown in the table, where n is the number of devices, and d_i and d_o are input and output dimensions, respectively. As seen, both methods somehow divide the memory footprint (i.e. saved weights) and the number of multiplications. Input splitting slightly increases the number of reductions because computing the partial sums is necessary on each device. Furthermore, output- and input-splitting methods have a communication overhead of $(n-1)d_i$ and $(n-1)d_o$, respectively. Depending on the size of the input and output, we can find the most optimum choice based on our device and communication.

As an example, in Figure 4.3, we run a series of dense layers on an RPi and their distributed versions on two RPis (in total four devices, with an initial sender and a final receiver). We cover a range of 512 to 16,384 in output sizes, and two input sizes, 7,680 (not a power of two) and 8,192 (a power of two). As seen, for the input size of 7,680 and large output sizes, we achieve super-linear speedups. This is because in these cases, slow off-chip storage (i.e., swap) is used. However, for the input of 8,192, the baseline DNN framework can optimize accesses and avoid swap activities by tiling. The baseline DNN framework optimizes the swap space accesses; however, it cannot always hide such costs for the input size of 7,680. Furthermore, if off-chip storage activities are not occurring in the baseline case, as seen, speedup values are less than the ideal value of two. This is because each distribution has a communication cost associated with it. We examine these costs and their impact on our distribution in Chapter 4. Figure 4.3 shows that the input-splitting method has mostly lower performance than output splitting. This is because the input-splitting method cannot apply activations locally. Therefore, input splitting cannot benefit from a reduced number of values to transfer, compared to output splitting. The reduction of values occurs because activation functions (such as ReLU), set every negative value (or close to zero values) to zero. Thus, in sum, fewer values are transferred after activation.

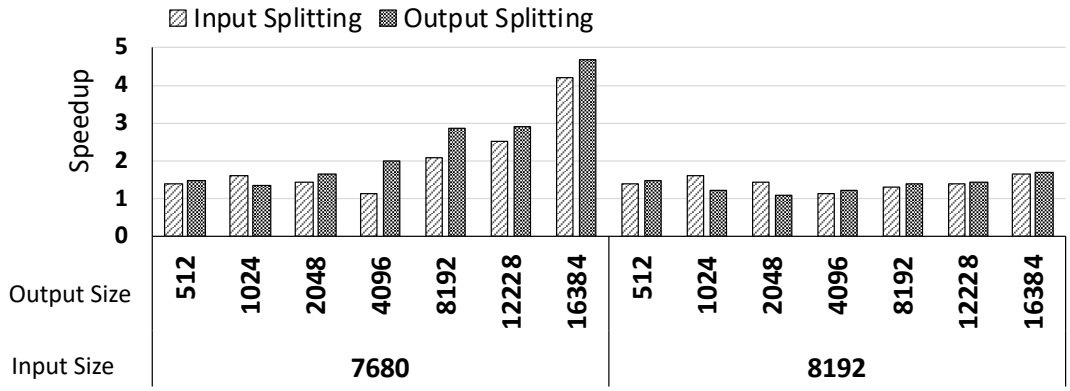


Figure 4.3: **Speedup Model Parallelism** – Performance of model-parallelism methods on two Raspberry Pis for fully-connected layers.

4.3 Model-Parallelism for Convolution Layers

In a convolution layer, each filter creates a channel in the output data. As Figure 2.2 illustrates, assume the dimensions of input, filters, and output are H_ixW_ixC_i, H_fxW_fxC_fxk, and H_oxW_oxC_o, respectively. The depth of the filters is defined by the depth of the input, or $C_f = C_i$. Here, without loss of generality, we assume square filters, $H_f = W_f = f$. The number of channels in the output is defined by the number of filters, or $C_o = k$. Each filter contains $C_i f^2$ weights that are set during training. *Per output element*, each filter performs $C_i f^2$ multiplications of its weights and input values, and one reduction operation. So, for k filters in a convolution layer, per output element, we perform $kC_i f^2$ multiplications and k reductions. Therefore, the total number of multiplications and reductions in a convolution layer for *all elements* is:

$$\begin{aligned} \text{Multiplications: } H_o W_o k C_i f^2 &\xrightarrow{\text{Same Padding}} H_i W_i k C_i f^2 \\ \text{Reductions: } H_o W_o k &\xrightarrow{\text{Same Padding}} H_i W_i k. \end{aligned} \tag{4.1}$$

For a single inference, the amount of communication is the sum of the number of input and output elements, or $(H_i W_i C_i) + (H_i W_i k) = (C_i + k)(H_i W_i)$.

In the rest of this section, we describe our specific methods of model parallelism for

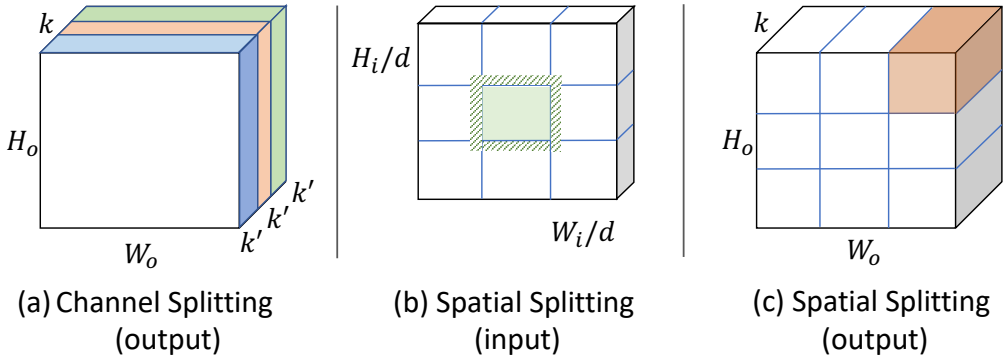


Figure 4.4: **Convolution Layer Channel and Spatial Splitting Methods** – (a) The output of channel splitting method with three splits. (b) The input of spatial splitting method with nine equal part with overlapped region highlighted for the middle part. (c) The output of spatial splitting method with nine equal part producing nine division in the output.

convolution layers. Since each method has advantages and disadvantages, Table 4.2 provides a detailed overview of the discussions in this section.

4.3.1 Channel Splitting

In channel splitting, each device calculates a non-overlapping set of channels in output. In other words, each device processes only k' filters that $k' \leq k$. Figure 4.4a shows an example output of this method with three devices. Since k' filter is processed per device, a total of $\lceil k/k' \rceil$ devices required. Each device needs not only its set of k' filters, but the entire input data. So, filters weights are divided across devices, or $k'C_i f^2$ per device. The total number of multiplications and reductions remains the same, and each device handles $\lceil k/k' \rceil^{-1}$ part. In the end, when every device is finished, their data is concatenated depth-wise, which is in $O(k)$. For the output, the total number of output elements to be transferred is $H_i W_i k$. We have the option to apply the activation function on each device or after the merging. In total, as shown in Table 4.2, communication overhead is $(\lceil k/k' \rceil C_i - 1)H_i W_i$, since we need to transmit a copy of the input to all devices.

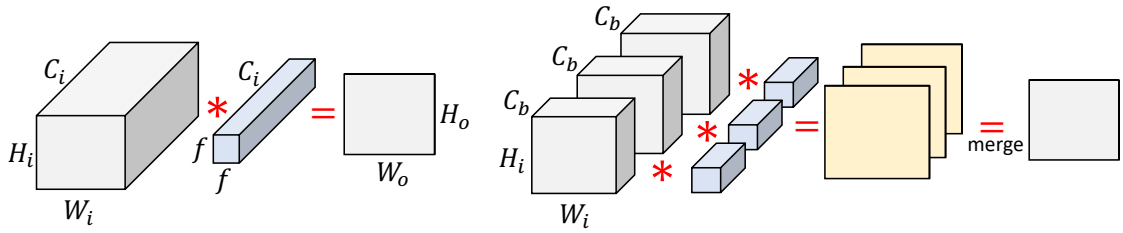
4.3.2 Spatial Splitting

Spatial splitting splits the input spatially, in the x- and y-axis. Assume that each split dimension is in d parts, so there are a total of d^2 parts[1], as shown in Figure 4.4b. Each part of the input is transmitted to a device. Furthermore, each region is extended for $\lfloor f/2 \rfloor$ more overlapping elements with neighboring parts, so that we can do convolution on the borders. Therefore, the number of input data elements to be transmitted per device is:

$$\lceil \frac{1}{d^2} \rceil H_i W_i C_i + 4\lfloor f/2 \rfloor (d^2 - d), \tag{4.2}$$

in which the first term represents the split input, and the second term represents the numbers of extra overlapping elements. Compared to channel splitting in which a copy of input is transmitted to all devices, spatial splitting only pays extra overhead for the overlapping elements. Since each device processes all filters, each needs a copy of all weights. Hence, the total number of filter elements to be transmitted is $d^2 k C_i f^2$. Note that this is a one-time cost for all inferences. The total number of multiplications and reductions is the same in total and each device processes only $1/d^2$. When the computation of each device is finished, their output is concatenated spatially. Similar to the previous method, the total number of output elements to be transferred is $H_i W_i k$. We have the option to apply the activation function either on each device or after the merging. As discussed, the communication overhead for spatial splitting is only for overlapping parts, which approximately is $4d^2\lfloor f/2 \rfloor (d^2 - d)$. Since the filter size is usually small, this overhead is not significant. Spatial splitting has another advantage, which is to generate a part of the output; we do not need to merge all the results. Therefore, in constructing a parallelized model while maintaining correctness, we can process a few convolution layers sequentially without merging their results back after every one layer.

[1] For simplicity, we divide each dimension to equal parts here. In our implementations, any number of divisions is possible.



(a) Convolution of one filter - Base case (b) Convolution of one filter - Filter splitting

Figure 4.5: **Convolution Layer Filter-Splitting Method** – (a) Baseline case of applying one filter. (b) Applying filter-splitting method on one filter shown in previous example.

4.3.3 Filter Splitting

In filter splitting, both input and filter are split channel-wise in batches of size C_b. Figure 4.5a illustrates the base case in the convolution of one filter, which produces a single channel in the output. Figure 4.5b illustrates the same filter divided into three parts with their corresponding input. Since there is a one-to-one correspondence between input and filter elements, each device computes a partial output. In the end, to create the final output, we sum all corresponding elements and apply the activation function. By denoting the input channel size as C_i, we need a total of $\lceil C_i/C_b \rceil$ devices. Since the input is split channel-wise, the total number of input element transfers is without an overhead, or $H_i W_i C_i$. Similarly, each device saves only its dedicated channels of all filters, so the memory footprint is also divided. But, since each device sends a partial output to the merging device, there is an overhead of $(k\lceil \frac{C_i}{C_b} \rceil - 1)(H_i W_i)$ for transmitting output elements compared to the baseline. To create the final output, we need to perform $k\lceil \frac{C_i}{C_b} \rceil$ reductions. The concatenation is in $O(C_i/C_b)$.

4.3.4 Other Splitting Methods

The three splitting methods discussed earlier, channel, filter, and spatial splitting are proposed in the context in which we have limited control over libraries or hardware implementations. In other words, the presented methods are applicable to any high-level machine learning framework (*e.g.*, TensorFlow [117]). This is because we used abstractions that are always

represented in such high-level frameworks. However, from a computation perspective, each convolution layer consists of six nested for loops (with the addition of one more loop for more than one input), which in reality could be split in several ways. In fact, although several of these new splitting methods could be represented in low-level programming languages (*e.g.*, C++), several others require custom hardware implementations. This is because our programming models have a compute-centric perspective rather than a data-centric perspective. For instance, Eyeriss [87] implements *row-stationary* splitting over its array of custom processing elements (PEs). However, deploying the row-stationary method using common tools and on conventional hardware requires a significant effort. These new splitting methods are out of the scope of this thesis and require more thorough analysis coupled with hardware constructs. Curious readers could check [118] for more information.

4.3.5 Methods Comparison

Table 4.3 presents a comparison of the described methods. Channel splitting has an overhead of copying the input, whereas filter splitting has to transmit partial sums. The impact of these differences on the performance is defined by the properties of a convolution layer. As illustrated in Figure 4.30, we run a convolution layer with the kernels 3x3, 5x5, 9x9, filter depths 128 and 512, and various input depths with 128x128 inputs. We distribute this layer on three RPis using the mentioned splitting methods (in total five devices, with an initial sender and a final receiver). Speedups are relative to single-device execution. We see that in the kernel 3x3 and filter depth 128, smaller input depths have no speedup. This is because the amount of computation per device after the distribution is small. However, for the large input depths, since the amount of computation after the distribution is more balanced, we see a speedup. Furthermore, in most cases, spatial splitting performs better. This is because, contrary to other methods, spatial splitting has less communication overhead. However, for larger 9x9 kernels, since the number of overlapping elements increases, the advantage of spatial splitting compared to other methods decreases.

4.4 Work Distribution

To apply both model and data parallelism, we first divide a DNN model on multiple devices by layers (or a group of layers) and create a processing pipeline. These layers process the input sequentially, and the output of each layer is dependent on the output of its previous layer(s). Thus, we must correctly maintain the dependence between layers. By utilizing this processing pipeline, we can increase the throughput of computation, while the latency for each computation remains the same. By applying model parallelism on top of this processing pipeline, we can further parallelize the computation of the bottleneck layers.

4.4.1 Why Distribution Helps Performance?

To understand why distributing and parallelizing DNN computations are necessary for edge devices, Figure 4.6 shows the memory usage and time to process an input (i.e., latency) of some layers in C3D and VGG16. As seen, fully-connected layers of both models have an extremely large memory footprint that causes long latency (in order of minutes, not shown). For fully-connected layers, this large memory footprint and low compute intensity activate the use of swap space. This behavior is true for almost all visual models since after extracting visual features using convolution layers, these models need to flatten the features for classification. Such conversion from visual features to categorical features, which are implemented with fully-connected layers, causes high memory usage with low computational density. We perform an experiment, the result of which is shown in Figure 4.7 that shows model parallelism has higher performance benefit than data parallelism for these layers. These results, with results in Figure 4.3, show how distribution achieves higher performance.

As discussed, applying model parallelism is necessary for layers with a large memory footprint, such as first fully-connected layers (Figure 4.6a),to bypass swap space usage. Convolution layers, on the other hand, have a much smaller memory footprint; but with a

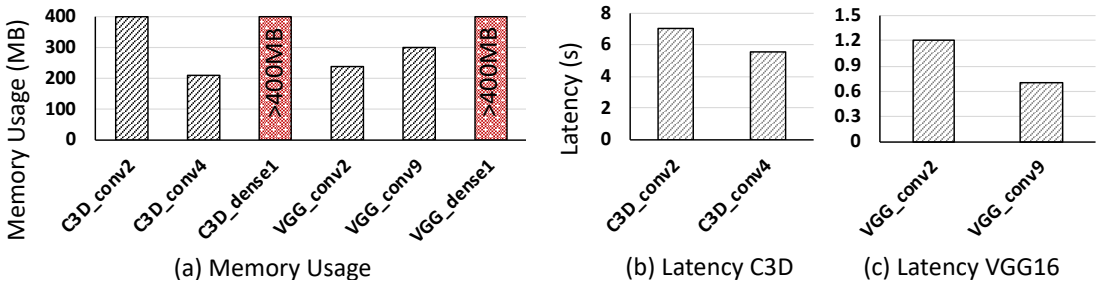


Figure 4.6: **Memory Usage and Latency of VGG16 and C3D** – Memory usage and latency of some layers in VGG16 and C3D models on a Raspberry Pi while performing a single inference.

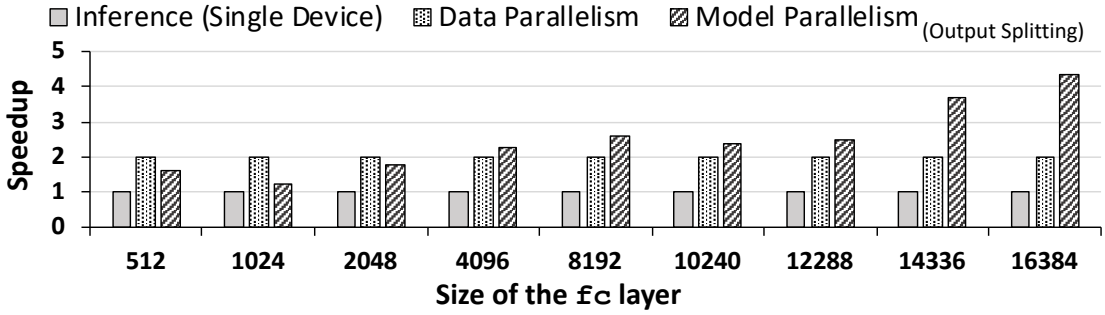


Figure 4.7: **Fully-Connected Layer Speedup for Model and Data Parallelism** – Performance speedup of model and data parallelism for fully-connected layers with different sizes and input size of 7,680 on two RPis.

few layers on a device, we will eventually exceed the available memory of the device and face the same issue. For convolution layers, it is also possible that the latency of a single layer is long and not suitable for real-time processing because of its large computation load. To this end, we present some examples in Figure 4.6 that show the latency of some convolution layers in VGG16 and C3D models. As illustrated, even for a single layer, the latencies are not suitable for real-time processing. In addition, as shown in Figure 4.30, we see that model parallelism is able to provide us with a performance benefit. Therefore, we see that model-parallelism methods for convolution layers are also useful for DNN models.

Note that most DNN models have more than 10 layers, and until now, as examples, we have shown only the statistics for single layers. The mentioned challenges are exacerbated with more layers. In summary, the total latency of executing the entire model on a single device is longer because *the latencies of all layers are accumulated because there*

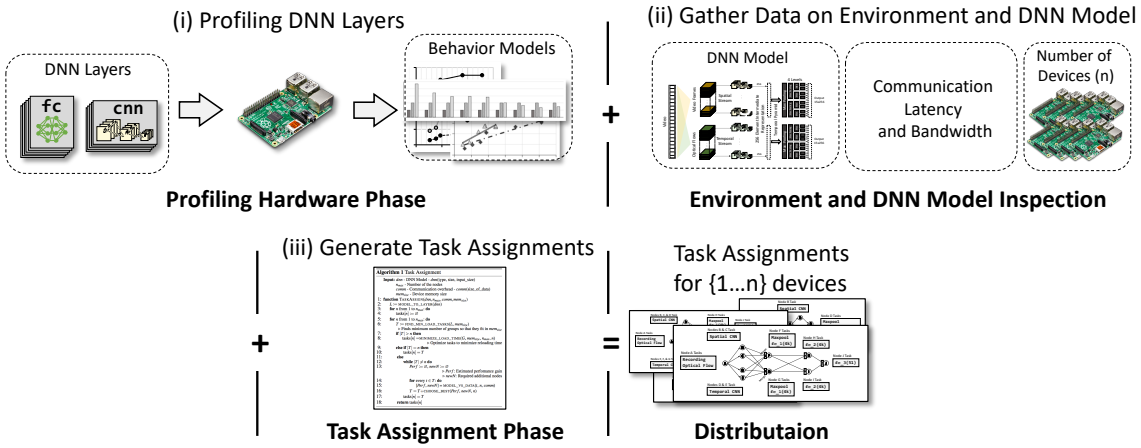


Figure 4.8: **Offline Work Assignment Procedure** – Steps for generating task assignments in offline work assignment with profiling.

are no parallelization opportunities. Model-parallelism methods help us to solve these challenges because they reduce the memory footprint and exploit more compute resources by introducing parallelism among devices.

4.4.2 Offline Assignment with Profiling

To find a close to optimal distribution for each DNN model, given the number of devices in the system, we devise a solution based on profiling. Our goal is to increase the number of performed inferences per second, or IPS. Profiling is necessary for understanding the performance benefits of data and model parallelism. In other words, we must consider whether assigning more than one task to any device will cause significant slowdown because of the limited memory resource or if data or model parallelism with its overheads, such as data transmits and merges, increases IPS. In our solution, Figure 4.8, first, for each layer, we profile execution times and memory usages of its original, model-parallelism, and data-parallelism variants. For each hardware system, the profiling is performed offline and only once for creating this data. Second, our solution takes the target DNN model, number of devices, and communication overhead (a regression model of latency based on the data size). Finally, using gathered data, we generate task assignments based on the flow of Algorithm 1.

Algorithm 1 Offline Task Assignment Algorithm.

```
1:  function TaskAssignment(dnn, n_max, comm, mem_size)
    Inputs: dnn: DNN model in form of layers[type, size, input_size, β]
            n_max: Maximum number of the devices
            comm: Communication overhead model (comm(size_data))
            mem_size: Device memory size
2:      L := extract_model_to_layers(dnn)
3:      for n from 1 to n_max: do
4:          tasks_final[n] := ∅
5:      for n from 1 to n_max: do
6:          TG, noFit := find_initial_taskGroup(L, mem_size)
7:          if sizeof(TG) > n then
8:              tasks[n] =combine_tasks(TG, mem_size, n_max, n)
9:          if sizeof(TG) = n then
10:             tasks[n] = TG
11:         if sizeof(TG) < n or noFit == True then
12:             while sizeof(TG) ≠ n do
13:                 task_variant := ∅
14:                 for every t ∈ TG: do
15:                     [task_variant] += profiled_variants(t, comm)
16:                 task_replaced, task_new = select_lowest([task_variant])
17:                 TG = TG − task_replaced + task_new
18:             tasks_final[n] = TG
19:         return tasks_final
```

In this algorithm, the function in Line 2 extracts the model input, *dnn*, into layers, L, while also accounting for buffering requirements (for videos, see subsection 4.5.2). Required extra buffers should be specified by the user in β. Because of the possibility that during execution some devices are inactive, busy, or have more than one input, we generate task assignments offline for all the possible number of devices (e.g., one, two, ..., total number of devices). For every number of devices, n, we create a dictionary of the node's name to its tasks, $tasks_{final}[n]$, and initialize it in Line 4 to the empty set. Then, from Line 5, we start a for loop for generating task assignments for the n number of devices. Since we generate all of the task assignments for any number of devices offline, our system can dynamically change the number of devices without the cost of computing a new assignment. To do so, first, the function in Line 6 generates an initial tasks group, TG, from L, such that every

entity in TG fits in mem_{size} of our devices. Basically, the function starts from the first layer while using the profiled data and creates a group of consecutive layers until they cannot fit in the mem_{size}, and then moves on for creating the next group. (If a single layer does not fit in the memory, $noFit$ flag is set for that entity in TG.) Then, based on the number of initial tasks groups, $sizeof(TG)$, the algorithm changes TG by adding or removing tasks until all n nodes are utilized, or $sizeof(TG) = n$. If $sizeof(TG) > n$, it means current tasks need more devices than what the system has, so we have to co-locate some tasks together and pay the overhead of task reloads. Hence, the function in Line 8 tries to combine two consecutive tasks (two tasks such that one produces data and the other consumes it directly) that together have the lowest memory consumption across all possible consecutive tasks and performs the process until the tasks fit on n devices. This is because lowest memory consumption is directly related to the lower reloading time of tasks to the memory. If $sizeof(TG) < n$ (or $noFit$ is set), the function in Line 15 uses the profiled data and the communication model, $comm$, to estimate the execution time of new task variants, $task_{variant}$, for all variants of the task, that is, original, model- and data-parallelism variants. Then, Line 16 chooses the variant with the lowest execution time across all possible variants for all tasks and outputs the to-be-replaced task ($task_{replaced}$) and the selected variant ($task_{new}$). Finally, Line 17 updates TG. This process continues in the while loop (Line 12) until we utilize all available devices, or $sizeof(TG) = n$. In this algorithm, since performance gain and communication overhead are estimations, optimality is not guaranteed. However, since task assignment is not in the critical path, we can fine-tune assignments before deployment (fine-tuning is not performed in our experiments).

4.4.3 Online Assignment with Monitoring

How can we find a near-optimal distribution for a given number of devices in runtime? The distributed system that we study is essentially a processing pipeline for the DNN model; each device processes a part of the computation and offloads the rest to the next devices.

Our goal is to find a distribution that achieves a near-the-optimal number of inferences per second (IPS) (higher is better) or the lowest latency (lower is better). Moreover, at the same time, the distribution should be able to improve itself in runtime (*i.e.*, online).

Form problem formulation, if we have $\mathcal{W}$ amount of work and n workers, our speedup compared to a single node case is:

$$\text{Speedup} = \frac{\mathcal{W} + \text{overhead}_{\text{single}}}{\mathcal{W}/n + \text{overhead}_{\text{pipeline}}}, \tag{4.3}$$

in which the $\text{overhead}_{\text{pipeline}}$ entails communication overhead ($\propto$ data size), and some fixed overhead such as the network set-up time between devices. Similarly, $\text{overhead}_{\text{single}}$ shows the overhead associated with the single-node execution, such as swap space activities. If the communication overhead dominates our distribution, and single device execution does not have significant overhead, we experience a slowdown after the distribution. Several examples of such layer configurations can be found in Figure 4.30. To avoid such scenarios, we need to (i) avoid unnecessary distribution to reduce the amount of communication overhead and (ii) associate enough work per node so the benefit of parallelizing exceeds the communication overhead. To do so, we merge less compute-intensive layers on a single node. As an online load-balancing technique, we also monitor idle nodes and combine the layers to increase the utilization of each node, thereby achieving a balanced pipeline. However, if the $\text{overhead}_{\text{single}}$ is significant, such as swap memory activities in fully-connected layers, in an acceptable range of $\text{overhead}_{\text{pipeline}}$, we experience speedups with distribution, as observed in Figure 4.3 and Figure 4.7.

Generating a Balanced Pipeline

To create a near-optimal distribution, the latency of each device should be similar to that of other devices. Thus, the amount of work per device, or $\mathcal{W}/n$, should be the same. Model parallelism helps us gain access to smaller granularities of work during distribution. On the other hand, data parallelism does not change the amount of work per device, but increases the throughput. With model parallelism, the throughput of a task increases, so the effective

latency seen by the next devices decreases. By considering these properties, to generate a distribution, first, we create a database with a mix of (i) regression models based on the amount of work and type of layers and (ii) profiled data from some layers and their split versions (similar to the results in subsection 4.3.5). Then, we study our given DNN model layer by layer. If the memory footprint is large and causes swap activities, for that layer, we have to first use model parallelism. After that, we try to group fewer compute-intensive and sequential layers to reduce the communication overhead. The grouping is done in a way that the average latency for processing an input on each device would be similar. After deploying such an initial distribution, we monitor the queue occupancy and latency of each device. With these gathered new data, we repeat the above steps to tune the distribution in runtime by creating a more balanced pipeline. Algorithm 2, in $O(n)$, summarizes these steps. The initial execution time (number of iterations of the procedure) until the system adjusts the performance depends on the complexity of the model. For the model in this paper, it takes less than 5 minutes, or around 25 iterations.

Algorithm 2 Heuristics for Online Task Assignment.

1: **procedure** GENERATEDISTRIBUTION
   Inputs: list of layers $\mathbb{L}$, #Nodes n
   mem_{size}: Memory size per node
   Regression models or profiling database, $\mathbb{D}$
   Outputs: dictionary of node IDs to a set of its tasks, $\mathbb{T}$
2: **Step1:** Check memory usage all layers in $\mathbb{L}$ using $\mathbb{D}$, if larger than mem_{size}, add that layer to the model parallelism list.
3: **Step2:** Using latency of layers in $\mathbb{D}$ and their split version, and by ensuring sequential dependency of layers, try to create groups of layers with the same latency. Create $\mathbb{T}$.
4: **Step3:** Deploy $\mathbb{T}$. Monitor queue occupancy and latency on each device. Goto Step2.

To give an unbiased view, the limitations of this approach are the following. First, for the initial deployment, there should be some initial measurements close to the size of each layer. Second, our procedure currently is evaluated in systems with the same type of devices.

Third, devices might lose some data points when the system is dynamically configured for a new distribution. Finally, we focus on DNN models for computer vision tasks. Note that although our discussions are about the execution of a single DNN model, one can extend our methods to multiple concurrent DNN models. However, the user needs to ensure that the system can handle the computation loads of concurrent models either by introducing more devices or designing reactive event-based systems.

4.4.4 Virtualized Execution

With a diverse set of hardware platforms and frameworks, virtualization could provide several benefits by decoupling hardware/software setup and reducing the programmer's effort. Moreover, cluster management services such as Kubernetes may provide several support for our systems. In fact, several such services have built in support for dynamic task assignment and monitoring. But, first, what is the overhead of virtualized execution?

Overhead of Virtualized Execution

Endeavors to design virtualization environment for each specific hardware platform to maximize performance are underway. But, virtualization itself has overhead. This overhead is caused by several translations for system calls and environment isolation. We evaluate the overhead of virtualized environments by executing DNN models inside and outside such an environment. We use Docker [119], a widely used virtualization tool in both academia and industry. Figure 4.9 shows the results of executing DNNs on RPi with/without Docker. As seen, the overhead is almost negligible, within 5%, in all cases. Contrary to popular belief about virtualization overhead, we do not observe a significant slowdown with virtualization.

Extension to Cluster Management

To improve the scalability and flexibility of our system, we integrate our framework with Docker virtualization service and Kubernetes cluster management service. Our previous

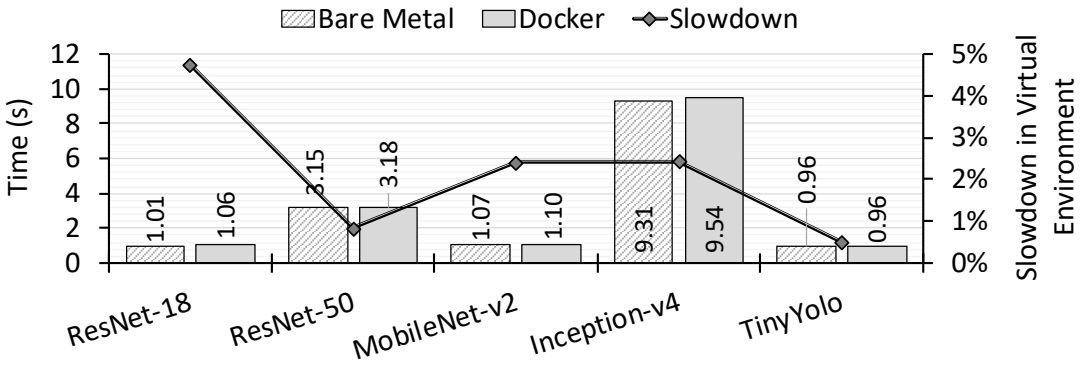


Figure 4.9: **Virtualization Overhead** – Time per inference on Bare Metal RPi and Docker-based RPi.

hand-written framework was not able to achieve real-time scalability because the framework could not adjust the service distribution in real time in an event when a certain service (i.e., task) becomes a bottleneck due to various reasons (i.e., network congestion, device might not be idle anymore). By adding Docker virtualization and Kubernetes management tool to our framework, we are able to scale up and scale down services in real time. We are also able to monitor the computation and memory usage of cluster nodes and manage the cluster effortlessly. In addition, with the virtualization features of Docker, we have the opportunity to extend our work to the heterogeneous systems (e.g., a mix of Raspberry Pis and Nvidia Jetson TX2) with adaption to different distribution setup.

Another issue we faced in our previous studies and framework is finding an optimal distribution for a given DNN model, which required us to fine-tune each distribution. To profile each unique distribution, we needed to put more than 30 minutes from code setup to execution. Moreover, since finding an optimal distribution is an NP-hard problem, it was nearly impossible to find the optimal distribution solution manually and with limited monitoring tools. Having a widely used framework (i.e., Kubernetes) that is able to efficiently monitor all performance metrics and test various distribution automatically creates the base infrastructure for extending our study, so that it can learn the optimal distribution. In the future, our system should be able to learn the optimal distribution by itself and use such distribution to optimize the real-time inference. Docker virtualization service and

Kubernetes management tool have been successfully integrated into our framework, which adds scalability and monitoring features during execution.

In details, we use up to 10 Raspberry Pi 3s that utilize our framework based on Docker and Kubernetes. Raspberry Pis are connected with a wireless router/switch. The following link provides a showcase of our system executing YOLO object recognition system: http://comparch.gatech.edu/hparch/sysml.

4.5 Dealing with Video Streams

Using deep learning to recognize what is happening in a video is action recognition. Recognizing human activities and classifying them (*i.e.*, action recognition) in videos is a challenging task. Although video streams are processed with similar models shown in section 2.2, but they require several new steps. Such DNN models, while performing still image classification, must also consider the temporal content in videos. To show these differences, we use the model of Ryoo et al. [4], which consists of two separate recognition streams, spatial and temporal convolution neural networks (CNNs), the outputs of which are combined in a temporal pyramid [120] and then fused in fully connected layers to produce predictions.

4.5.1 Overview of Two-Stream Video Recognition Model

Spatial Stream CNN

The spatial stream, similar to image recognition models that classify raw still frames from the video (*i.e.*, images), is implemented with `conv` layers. This model, as input, takes a frame of size 16x12x3 (in RGB) and processes it with three `conv` layers, each with 256 filters, the kernel sizes of which are 5x5, 3x3, and 3x3, respectively. Then, features of each frame are summarized in a 256-element vector. Since this stream processes still images, for training, we can use any representative dataset, such as ImageNet [46], by adding a dummy output dense layer.

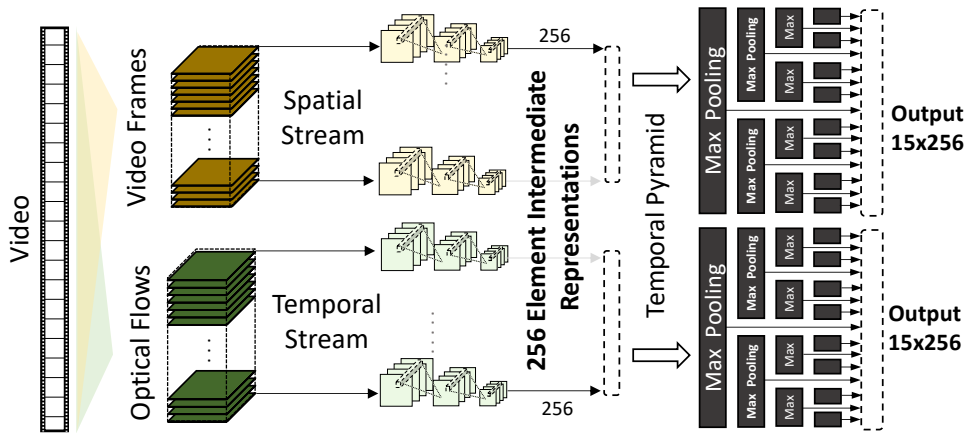


Figure 4.10: **A Video Recognition Model** – The temporal pyramid generation from spatial and temporal CNNs.

Temporal Stream CNN

The temporal stream takes optical flow as input, which explicitly describes the motion between video frames (we use Färenback [121] algorithm). In other words, for every pixel at a position (u_t, v_t) at time t, the algorithm finds a displacement vector $\mathbf{d}_t$ for each pair of consecutive frames, or $\mathbf{d}_t = (d_t^x, d_t^y) = (u_{t+\Delta t} - u_t, v_{t+\Delta t} - v_t)$. We compute the optical flow for 10 consecutive frames and stack their (d_t^x, d_t^y) to create an input with the size of 16x12x20. Subsequently, the data is processed with three `conv` layers, each with 256 filters, the kernel sizes of which are 5x5, 3x3, and 3x3, respectively. Finally, the features are summarized in a 256-element vector. By adding a dummy output dense layer, we can train the temporal stream with any video dataset, such as HMDB [122].

Temporal Pyramid

To generate a single representation from the two streams, a single spatio-temporal pyramid [120] is generated for each video. Figure 4.10 depicts the steps of generating a four-level temporal pyramid from a video. Such a pyramid structure of `maxpool` layers creates an output with a fixed size that is agnostic to the duration of videos. For each stream, 15 `maxpool` layers with different input ranges generate a 15x256 output. Finally, the data with size 2x15x256 is processed by two `fc` layers with sizes of 8192, and an `fc` layer with the

size of 51 outputs HMDB classes.

4.5.2 Sliding Window

The described action recognition model processes a whole video for each inference. However, in reality, the frames of a video are generated by a camera (30 FPS). To adapt a model for real-time processing, we propose the use of a sliding window over the input and intermediate data, whenever needed, while distributing the model. For instance, the temporal stream accepts an input of optical flows from 10 consecutive frames, so a sliding window of size 10 over the recent inputs is required. In a sliding window, whenever new data arrives, we remove the oldest data and add the new data to the sliding window. Note that to order arriving data, a unique tag is assigned to each raw data during recording time. Figure 4.11 illustrates this point with an example of eight devices in a system. The recorder device keeps a sliding window of size 10 to supply the data, while the devices that process spatial and temporal streams do not have a sliding window buffer. On the other hand, since the temporal pyramid calculation requires a spatial data of 15 frames and temporal data of 25 frames, the last device keeps two sliding window buffers with different sizes. We can extend the sliding window concept to other models that have a dependency between their inputs to create a continuous data flow. Furthermore, the sliding window is required to enable data and model parallelism. This is because a device needs to order its input data while buffering arrived unordered data.

As shown for our example action recognition model, we can extend CNNs for still images to video by utilizing a sliding window. Since this implementation uses memory as buffer, it does not require extra computations. However, the memory footprint of the model per device will increase depending on the sliding window size.

4.6 Experimental Results

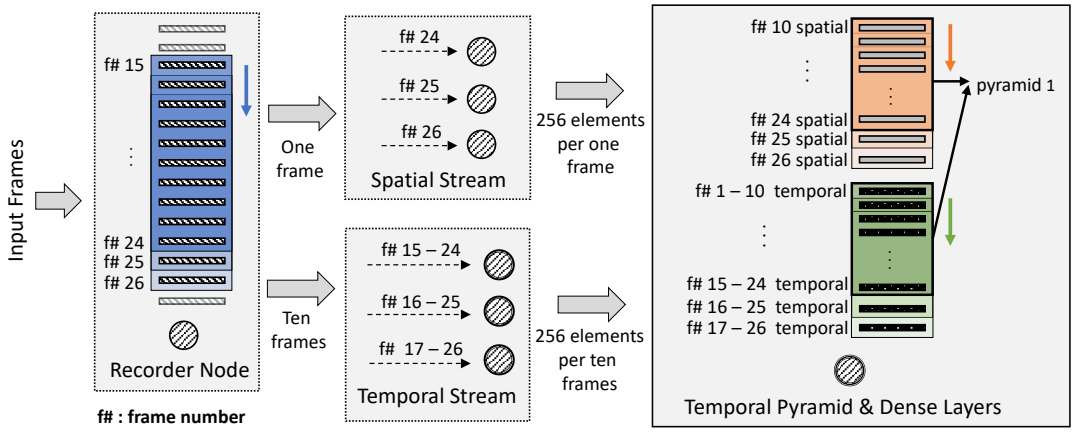
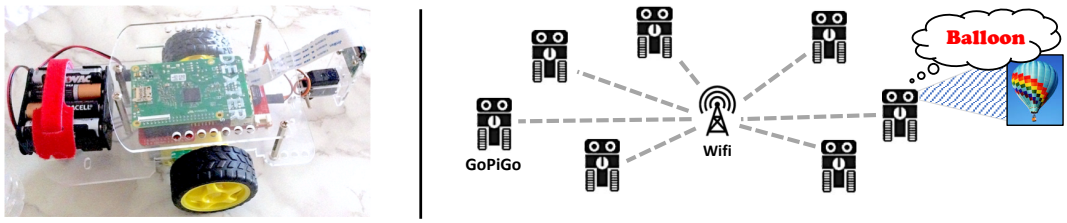


Figure 4.11: **Sliding Window Example** – Sliding window for an example system of eight devices. While some tasks require sliding window, with different sizes, others may not need it.

4.6.1 Evaluation Setup

We evaluate our methods on a distributed system with Raspberry Pi 3s (RPis) [41]. Table 4.4 presents the specifications of an RPi. Additionally, we use a Raspberry-Pi-based robot to further illustrate challenges with resource-constrained robots, shown in Figure 4.12. To provide a comparison reference, we also execute DNN models on two localized implementations: (i) Nvidia Jetson TX2 [123], the specifications of which are in Table 4.6. TX2 is a high-performance embedded platform with both a CPU and GPU with 8 GB DDR4. In contrast, RPis is an edge device with no GPU and less than 1 GB DDR2. (ii) A high-performance (HPC) machine (Table 4.5). For our implementations, we created a software stack with Docker containers. We use Keras 2.1 [124] with the TensorFlow backend (version 1.5) [117].



(a) GoPiGo Robot (b) Our Distributed Robot System

Figure 4.12: **GoPiGo Raspberry-Pi-Based Robot** – (a) GoPiGo robot and (b) distributed robot system.

For RPC calls and serialization, we use Apache Avro [125]. We use an IP table file to assign tasks to each device. A local WiFi network with the measured bandwidth of 94.1 Mbps and a measured client-to-client latency of 0.3 ms for 64 B is used. All trained weights are loaded to each Pi's storage (16 GB storage in our system), so each Pi can be assigned to execute any part of a layer. Note that each Pi has an SD card storage, for storing the weights, which is relatively inexpensive compared to the main memory. If local storage is limited, the assigned weight can also be shared in the network from a network-storage filesystem. This approach makes a tradeoff between how fast the switching time between different models can be and per-device storage usage

4.6.2 Offline Assignments

Single Device/Robot

Since a single robot has limited memory, it usually cannot handle the execution of all the tasks efficiently because for performing any computation, data should be loaded to memory

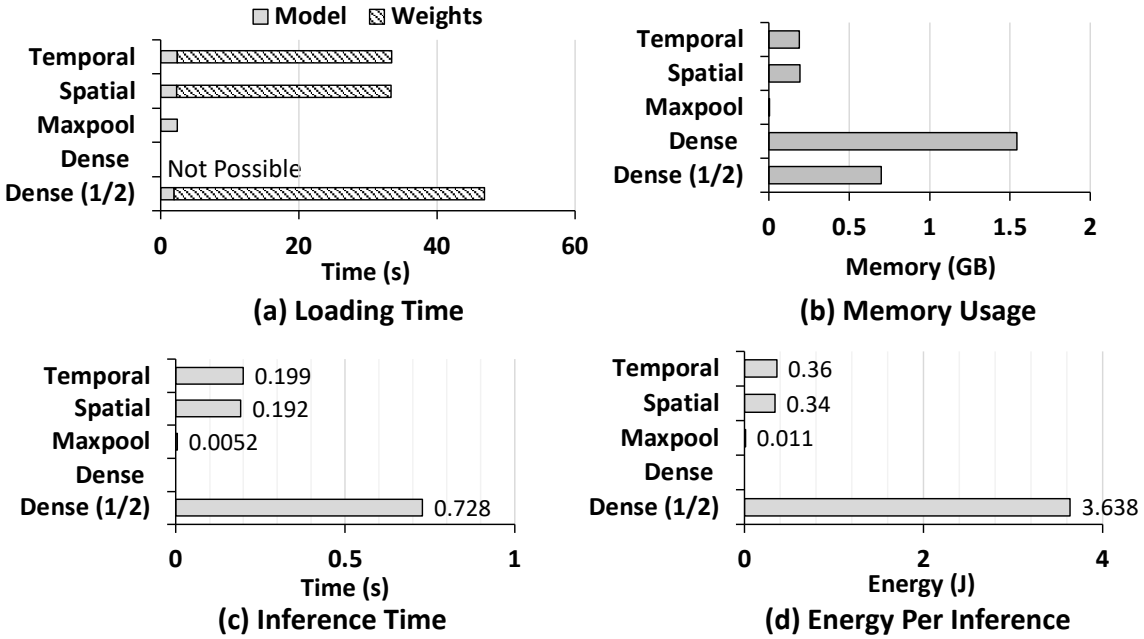


Figure 4.13: **Action Recognition Model on a Raspberry Pi** – (a) Loading time, (b) memory usage, (c) time per inference, and (d) energy per inference of general tasks in action recognition on a Raspberry Pi.

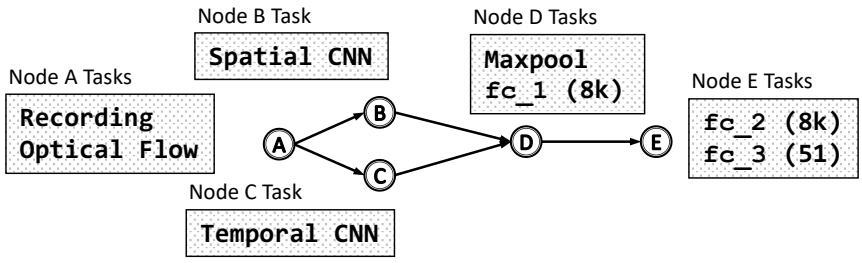
from storage. Figure 4.13a and b show the loading time and memory usage of general tasks in the action recognition model. The memory requirement of dense layers is larger than 1 GB, so a single robot needs to store and load intermediate states (*i.e.*, activations of a layer) to its storage, which incurs high delays. To gain insight, we even try a dense layer with half-sized dimensions of the original one, with 15% lower accuracy. Figure 4.13 shows that, in this case, even with a negligible computation time, the overhead of loading each task is high for real-time processing. Even when assuming zero loading time, as in Figure 4.13c and d depict for energy and inference time, the inference time of the half-sized `fc` layer is more than 0.7 seconds, while its energy per inference is 10x larger than that of spatial/temporal streams. Hence, in such an implementation, we still cannot process data in real time.

Action Recognition Model Distributions

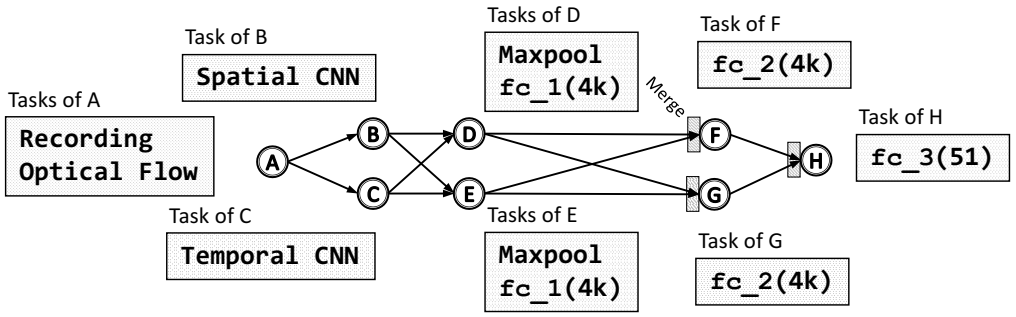
In the action recognition model, the recording robot also computes optical flow, the computation of which is not heavy (*e.g.*, 4 ms for 100 frames using the method in [121]). Each robot manages a sliding window buffer, explained in section 4.5, the size of which is dependent on the model and data parallelism of the previous robot and the input of the next robot. As discussed in the previous section, a single robot is unable to process data efficiently in real time. Hence, for demonstration, we perform distributed perception utilizing various systems, as shown in Figure 4.14, while measuring IPS, energy consumption, and end-to-end latency (Figure 4.15, Figure 4.16, and Figure 4.17, respectively)[2]. Our first system has five robots, Figure 4.14a, for which the final *fc* layers are distributed. Note that the systems with fewer than five robots are bounded by reloading time, and do not experience significant improvements in performance.

From eight robots, Figure 4.14b, our method performs model parallelism on both *fc* layers, creating two 4,096 `fc` layers per each layer. Furthermore, we are able to achieve 4.6x

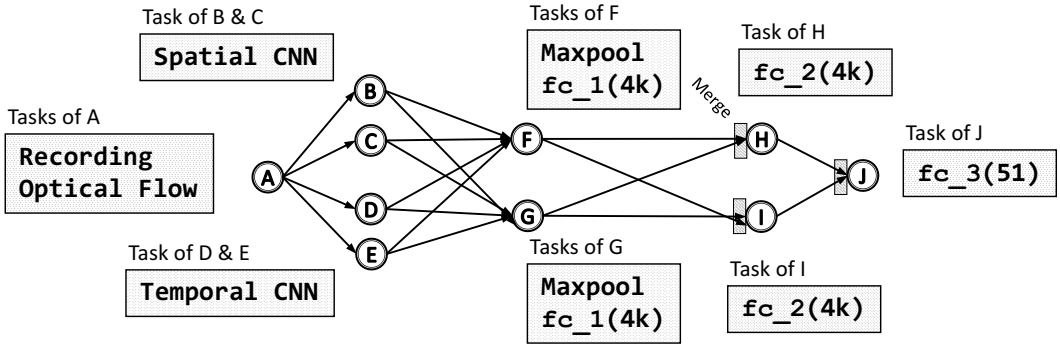
[2]We evaluate these experiments and make the source code publicly available in this artifact [126].



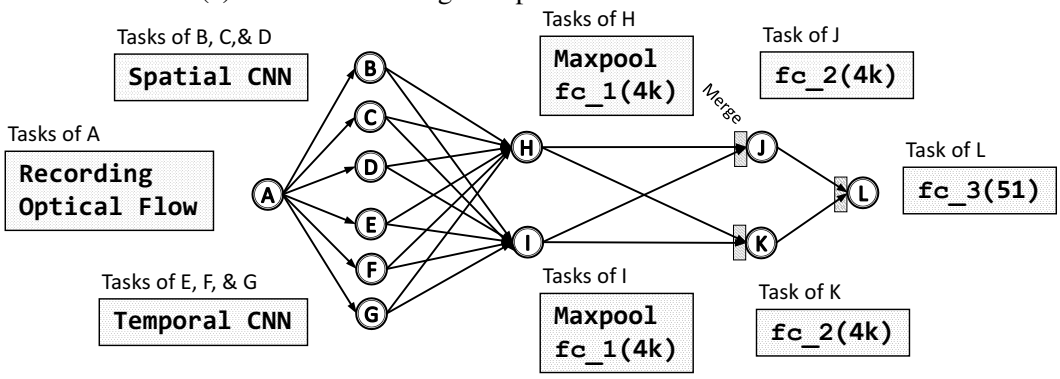
(a) Five robots: Exploiting model parallelism for `fc` layers.



(b) Eight robots: Exploiting model parallelism for *each* `fc` layer.



(c) 10 robots: Adding data parallelism for the two streams.



(d) 12 robots: Adding more data parallelism for the two streams.

Figure 4.14: **System Architectures of Action Recognition** – Illustrating task assignments for five, eight, ten and 12 devices/robots

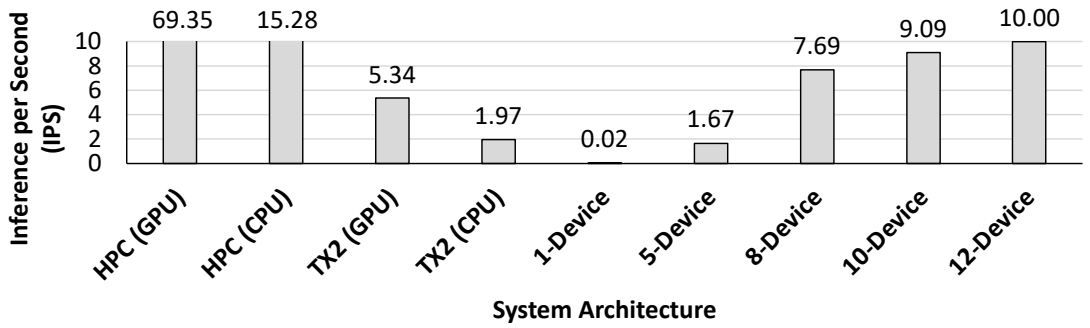


Figure 4.15: **Inference per Second** – Measured inference per second (IPS).

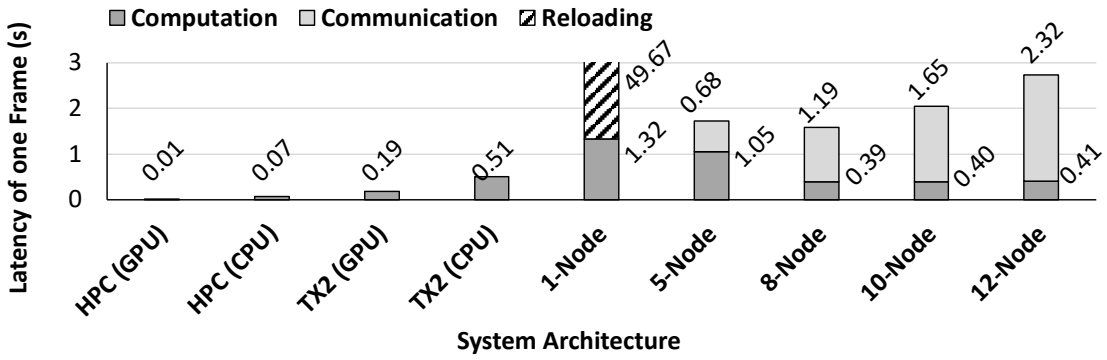
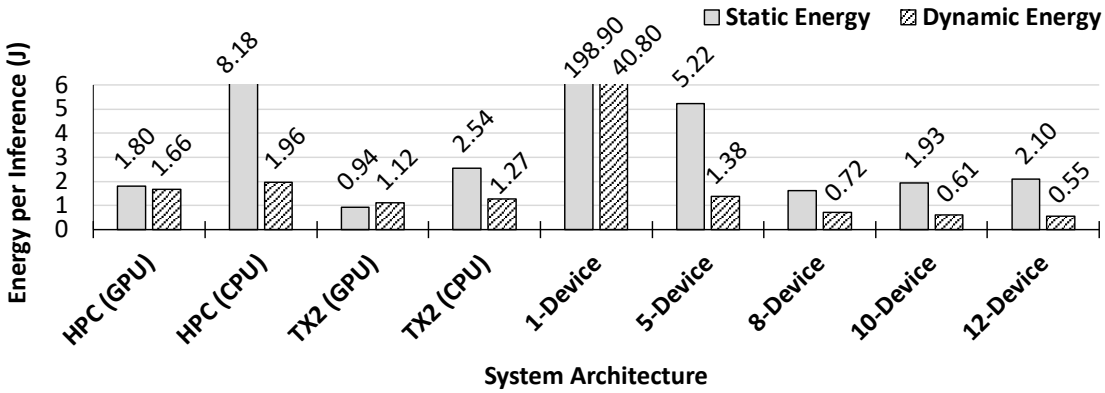


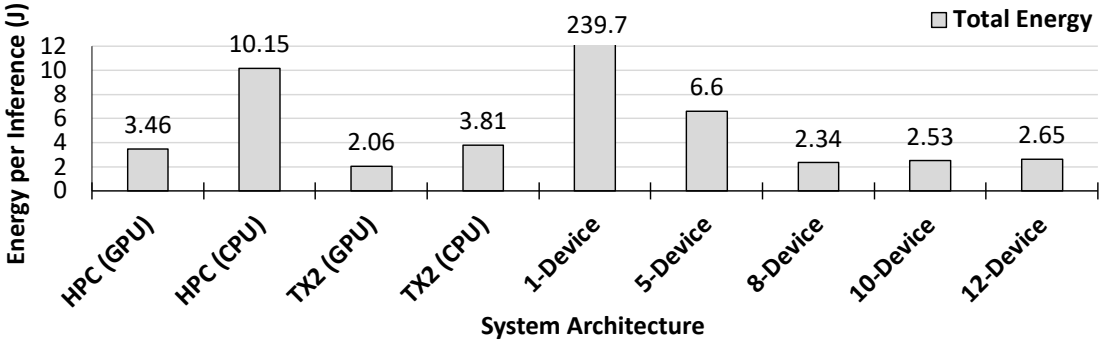
Figure 4.16: **Latency** – Measured end-to-end latency of one frame.

improvement in the performance and exceed the performance of TX2, shown in Figure 4.15. In the 10-robot system, two more robots process temporal and spatial streams exploiting data parallelism, illustrated in Figure 4.14c. New frames and optical flows are assigned in a round-robin fashion to two robots (of each stream) and are ordered using tags in subsequent robots. Finally, in the 12-robot system, more robots are assigned to process temporal and spatial streams with data parallelism. In summary, in comparison with a single robot, we gain up to 90x energy savings and a speedup of 500x for IPS. As Figure 4.15 and Figure 4.16 depict, although increasing the number of devices in a system also increases the latency notably, we observe a performance gain in IPS with a higher number of devices. This is because in both data and model parallelism, the systems hide latency by distributing or parallelizing tasks.

For the larger number of robots, we achieve not only similar energy consumption with TX2 but also save energy in comparison with the HPC machine. Figure 4.17b depicts that,



(a) Measured static and dynamic energy consumption per inference.



(b) Measured total energy consumption per inference.

Figure 4.17: **Energy** – Energy consumption per inference.

except for the TX2 with GPU, the energy consumption per inference (i.e., $^{\text{Watt}}/_{\text{performance}}$) of systems with more than five robots is always better than in other cases (up to 4.3x and an average of 1.5x). Note that in our evaluations, the power consumption of the robot systems is inclined to higher energy consumption because (i) in comparison with TX2, since each robot's Raspberry-Pi is on a development board, it has several unnecessary peripherals, the energy consumption of which increases significantly with more robots, which is shown in static energy; (ii) TX2 is a low-power design with power gating capabilities that gates three cores if not needed, but robots do not have such capabilities; and (iii) the energy consumption of the robot systems also includes the energy for communication between the devices and the wasted energy of powering an idle core during data transmission.

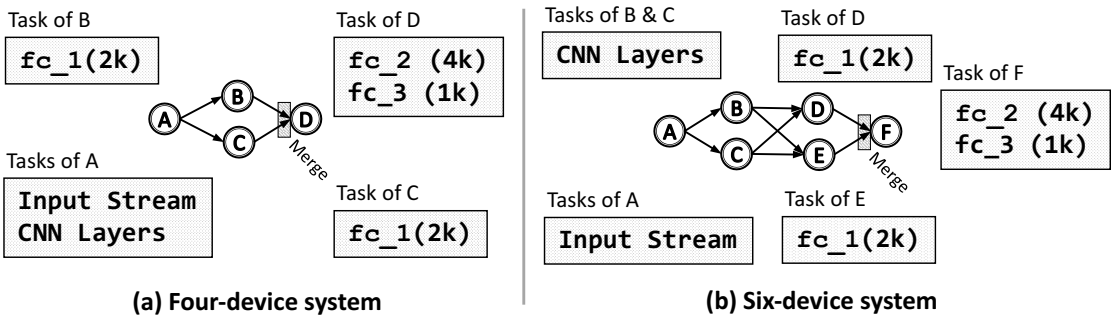


Figure 4.18: **System Architectures for AlexNet.**

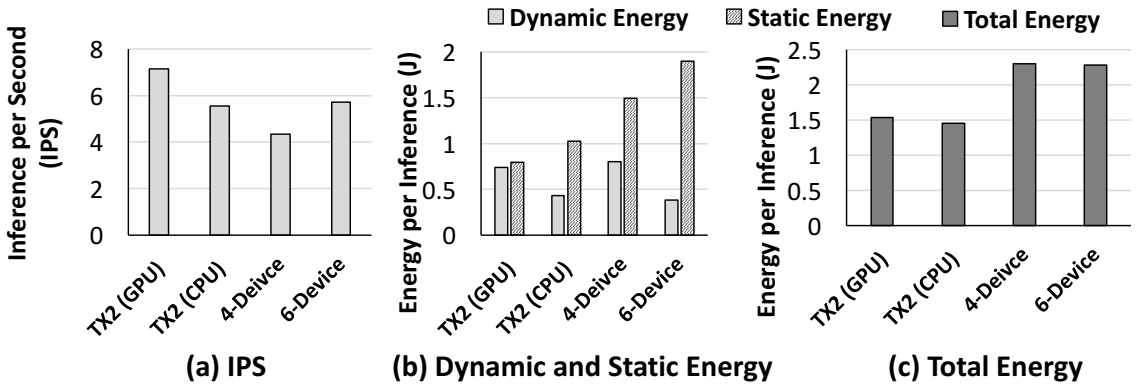


Figure 4.19: **AlexNet System Measured Statistics** – Measured IPS (a), static and dynamic energy consumption (b), and total energy consumption (c).

Image Recognition Models

We apply our method to two popular image recognition models, described in section 2.2. For AlexNet, Figure 4.18a and b display the generated tasks for four- and six-robot systems, respectively. While in the four-robot system, model parallelism is performed on the `fc_1` layer, in the six-robot system, additional data parallelism is performed on `conv` layers. We implement both systems and measure their performance and energy consumption, shown in Figure 4.19. Figure 4.19a depicts a performance increment by increasing the number of devices in a system. In fact, the achieved performance of the six-robot system is similar to the TX2 with CPU, and 30% worse than the TX2 with GPU. Furthermore, as discussed in the previous section, Figure 4.19b shows that most of the energy consumption of the Raspberry-Pi-based robots is because of the static energy consumption.

VGG16 (Figure 2.6), in comparison with AlexNet, is more computationally inten-

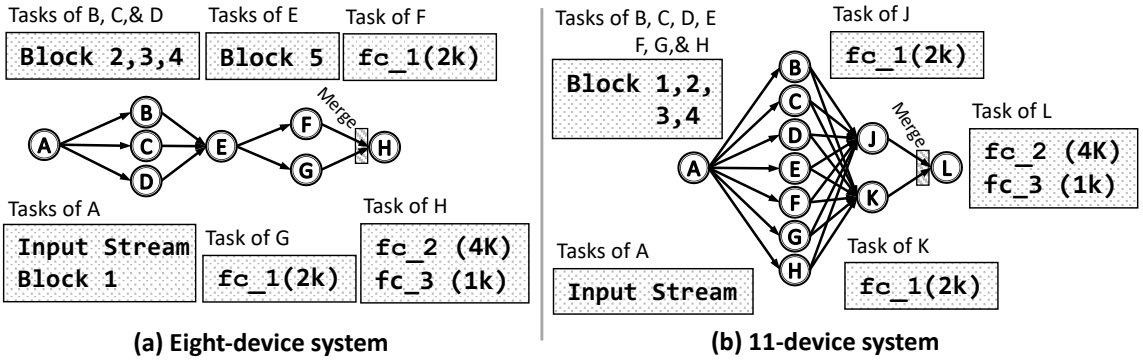


Figure 4.20: **System Architectures for VGG16.**

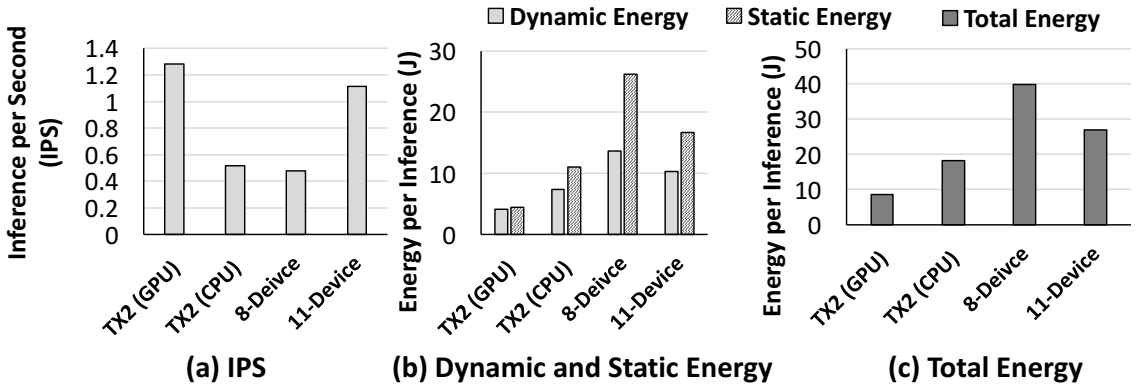


Figure 4.21: **VGG16 System Measured Statistics** – Measured IPS (a), static and dynamic energy consumption (b), and total energy consumption (c).

sive [127]. To distribute the model, our method divides the VGG16 model to several blocks of sequential `conv` layers. Figure 4.20a and Figure 4.20b depict the outcome of task assignment for VGG16 with eight and 11 robots, respectively. Our method for `fc_1`, since its input size is large, performs model parallelism, while for `fc_2` and `fc_3`, since their computations are not a bottleneck, it assigns them to a single robot. We measure the performance and energy consumption of both systems and the TX2, shown in Figure 4.21. When the number of robots increases from eight to 11, we achieve 2.3x better performance by reassigning all `conv` blocks to a robot and performing more optimal data parallelism. In fact, compared to the TX2 with GPU, the 11-robot system achieves comparable IPS (15% degradation).

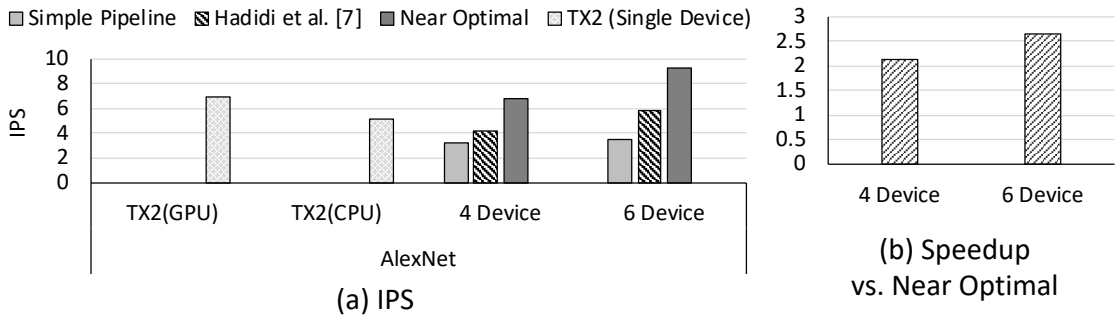


Figure 4.22: **AlexNet Online System Measurements** – AlexNet results on distributed systems compared with Jetson TX2.

4.6.3 Online Assignments

For online assignment system, after handshaking, which takes less than one minute, the system is ready. An initial task distribution in deployed in the beginning. During runtime, each device reports its latency and request queue occupancy. By collecting such stats, we are able to find bottleneck devices in our pipeline and create a more balanced pipeline, as Algorithm 2 describes.

AlexNet & VGG16

We deploy AlexNet and VGG16 models, including the last fully-connected layers, on various distributed systems. Since the first fully-connected layer in AlexNet faces a limited memory issue on an RPi, all of our distributions perform output splitting for this layer. The rest of the convolution layers are allocated to idle devices. Our two near-optimal systems have four and six devices and achieve higher than 2× speedups compared to distributed systems with a simple pipeline, Figure 4.22b. Because AlexNet layers all have low computation requirements, we could not get more benefit by distributing the computations. Figure 4.22a presents a more detailed performance measurement for AlexNet. Compared with TX2 with a GPU and CPU, the six-device distribution has a higher performance.

The VGG16 model consists of more computationally intensive layers compared to the layers of AlexNet. Therefore, we use eight and 10 devices for distribution to achieve

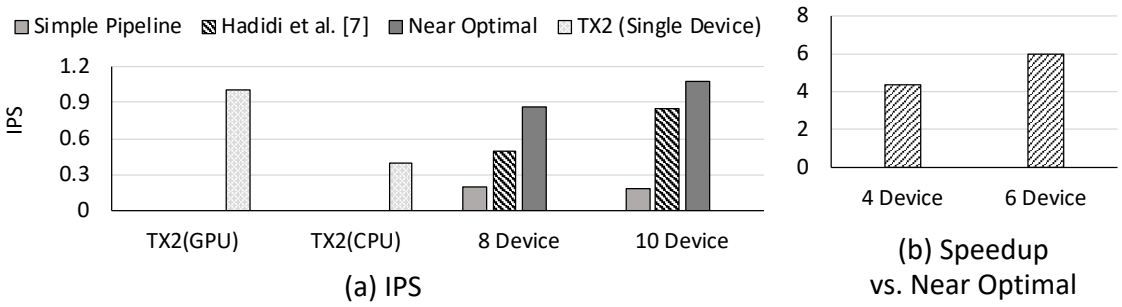


Figure 4.23: **VGG16 Online System Measurements** – VGG16 results on distributed systems compared with Jetson TX2.

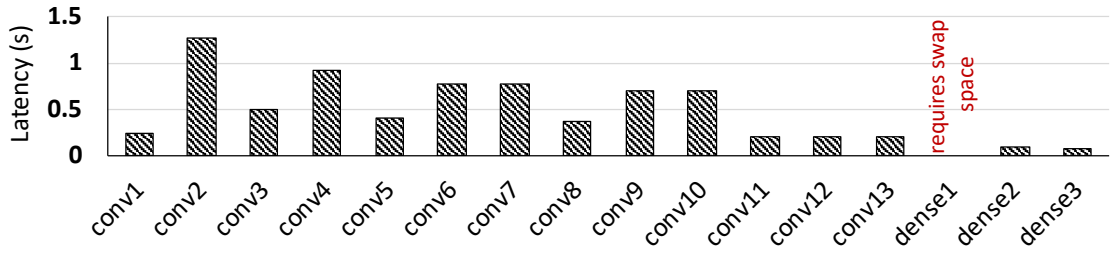


Figure 4.24: **VGG16 layer-wise latency** – Layer-wise latency on a Raspberry Pi (single inference).

up to 6× speedup compared to the simple pipelining scenario, as shown in Figure 4.23b. Moreover, as shown in Figure 4.23a, with more performance details, both of our distributions have higher performance than TX2 CPU. Our 10-device distribution also achieves similar performance to TX2 GPU. It is worth noting that both of our near-optimal distributions have higher performance than the TX2 CPU and simple pipelining scenario. Similar to AlexNet, since we include the first fully-connected layer, all of our distributions perform output splitting for this layer. For other layers, to gain a better insight, in Figure 4.24, we measured the layer-wise latency of VGG16 layers that are executed on RPi. Except for the first fully-connected layer, we are able to run all other layers on a single RPi. But, some layers have extremely long latencies, so we are bounded by such layers in our simple pipelining scenario (e.g., second convolution layer). On the other hand, in our eight- and 10-device systems with the near-optimal distribution, we bypass this bottleneck by using the model-parallelism methods for convolution layers, that are proposed in section 2.1.

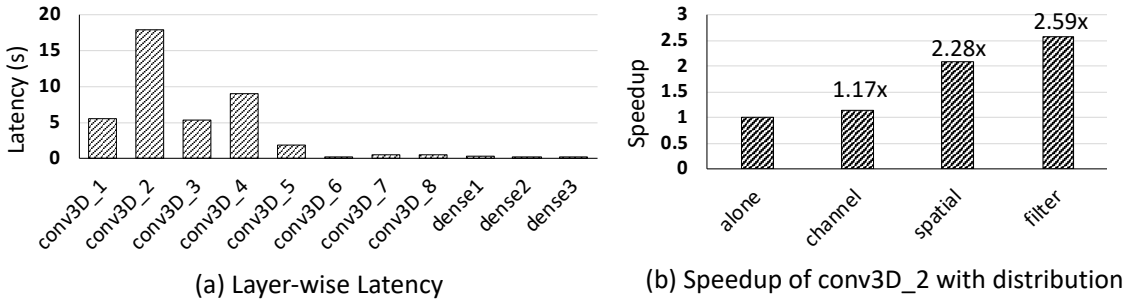


Figure 4.25: **C3D Model Deployment Measurements I** – (a) C3D layer-wise latency of a single inference. (b) Achieved performance after applying model-parallelism methods on the heaviest layer.

C3D

The C3D model, as discussed in section 2.2, incorporates 3D convolution layers. To understand this model behavior, we analyze the layer-by-layer latency of C3D models on the RPi in Figure 4.25a. As shown, the first layers of C3D are quite heavy for IoT devices. For instance, the latency of the second convolution layer is 18 seconds. This high latency is caused by the high computational demands of 3D convolutions. Model-Parallelism methods for convolution layers are particularly useful in distributing this among all devices. We apply our three methods of model-parallelism on three devices for the second (heaviest) convolution layer. As seen, we attain up to a 2.6× speedup by using three devices for this layer. Note that the spatial- and filter-splitting methods achieve higher performance than the channel-splitting method. This is because the size of the input is large, and therefore methods such as channel splitting, which does not divide the input, have a high overhead for communicating the copies to all devices, whereas, both spatial- and filter-splitting methods have a lower overhead due to the split input.

To get an estimation of the overall performance of C3D, we select the heaviest layers of the C3D model, (`conv3D_2`, `conv3D_2`, and `conv3D_4`) and deploy them on a distributed system using our heuristics. The first system, our baseline, is simply the sequential execution. By introducing extra devices, our heuristics split the computations of `conv3D_2`, similar to Figure 4.25b. The results for both filter- and channel-splitting methods for four and

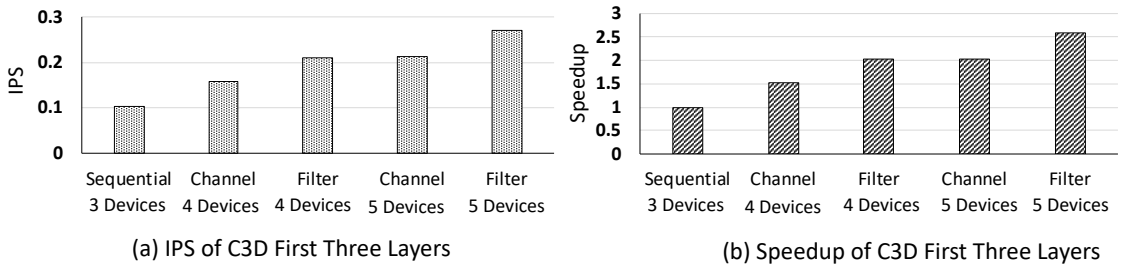


Figure 4.26: **C3D Model Deployment Measurements II** – C3D layer-wise first three layer deployment on various systems.

five devices are shown in Figure 4.26. As shown, with a higher number of devices, the performance gain also increases. In all variations, the filter-splitting method, as observed in Figure 4.30 and discussed in the previous paragraph, achieves higher performance than channel splitting.

ResNet50 & Xception

The ResNet50 and Xception models have similar building blocks, as shown in Figure 2.7, Figure 2.8, and Figure 2.9. For practical reasons of the limited number of devices, we choose to experiment with Xception. Since the building blocks of both models are similar, our observations are extendable to ResNet models as well. We measure the layer-wise latency of layers in Xception during single-batch inferencing, shown in Figure 4.31. As seen, in comparison with AlexNet and VGG16, for which the final fully-connected layers were the most compute-intensive and resource-hungry layers, in Xception, some convolution layers are more compute intensive and resource hungry. To better understand the aggregated processing time for Xception, we measured the total latency of different blocks in Xception (as shown in Figure 2.9), when they are executed on a single RPi. Figure 4.27 depicts the measured latencies. As seen, block C has the longest latency among other blocks. Since Xception is a large model, we deployed only one block in our system. We chose the heaviest block (i.e., block C) and deployed it on three different systems, shown in Figure 4.29.

The system shown in Figure 4.29a shows a simple sequential distribution, in which each

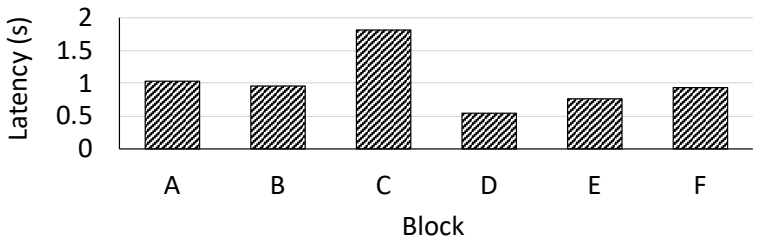


Figure 4.27: **Experiments on Xception Blocks** – Execution latency of Xception per block on a RPi (single inference).

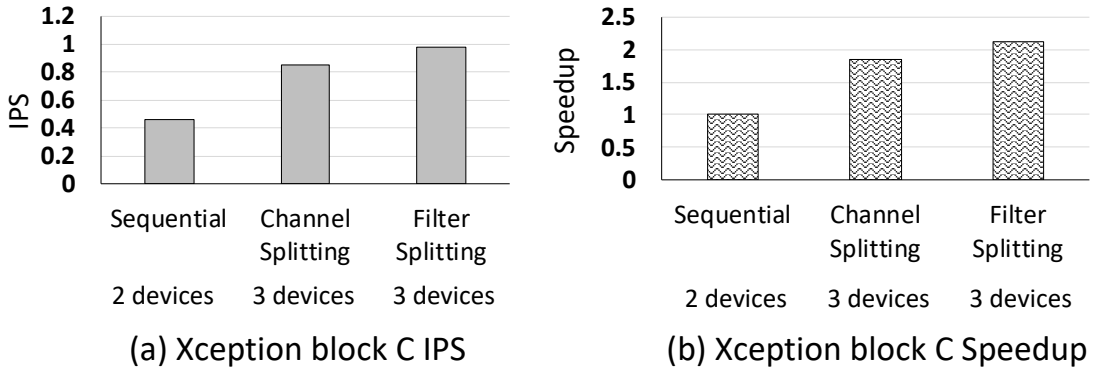


Figure 4.28: **Experiments on Xception Block C** – (a) IPS and (b) performance speedup for the systems shown in Figure 4.29, consisting of multiple RPis.

device processes a layer. Figure 4.29b shows a system that uses channel-splitting method for the heaviest convolution layer in block C. Similarly, Figure 4.29c illustrates a system in which the heaviest convolution layer in the block C is distributed using the filter-splitting method. The performance comparisons of these systems are shown in Figure 4.28. As seen, by including another device, our system can achieve up to a 2× speedup.

Figure 4.29, also depicts the queue occupancy of the devices that is extracted from our monitoring tools. The histograms in the figure show the queue occupancy of the devices. Note that queue size per device is limited to 10 requests. As seen, in Figure 4.29a, the queue of device B is always full. Therefore, our heuristics apply splitting to the work that is performed in device B. Figures Figure 4.29b and c show such splitting for the channel and filter splitting methods, respectively. Although we still see a close-to-full occupancy for devices B and C, which perform the split job, we observe that device A occupancy has

shifted to the right. This shows that our method was successful in creating a more balanced work distribution, but did not have enough available devices to create the best distribution. Note that, as discussed, our heuristics have access to a database of similar experiments that are done in Figure 4.30. Therefore, it does not need to perform both splittings to find the best performing one. Here, we are showing both as an example.

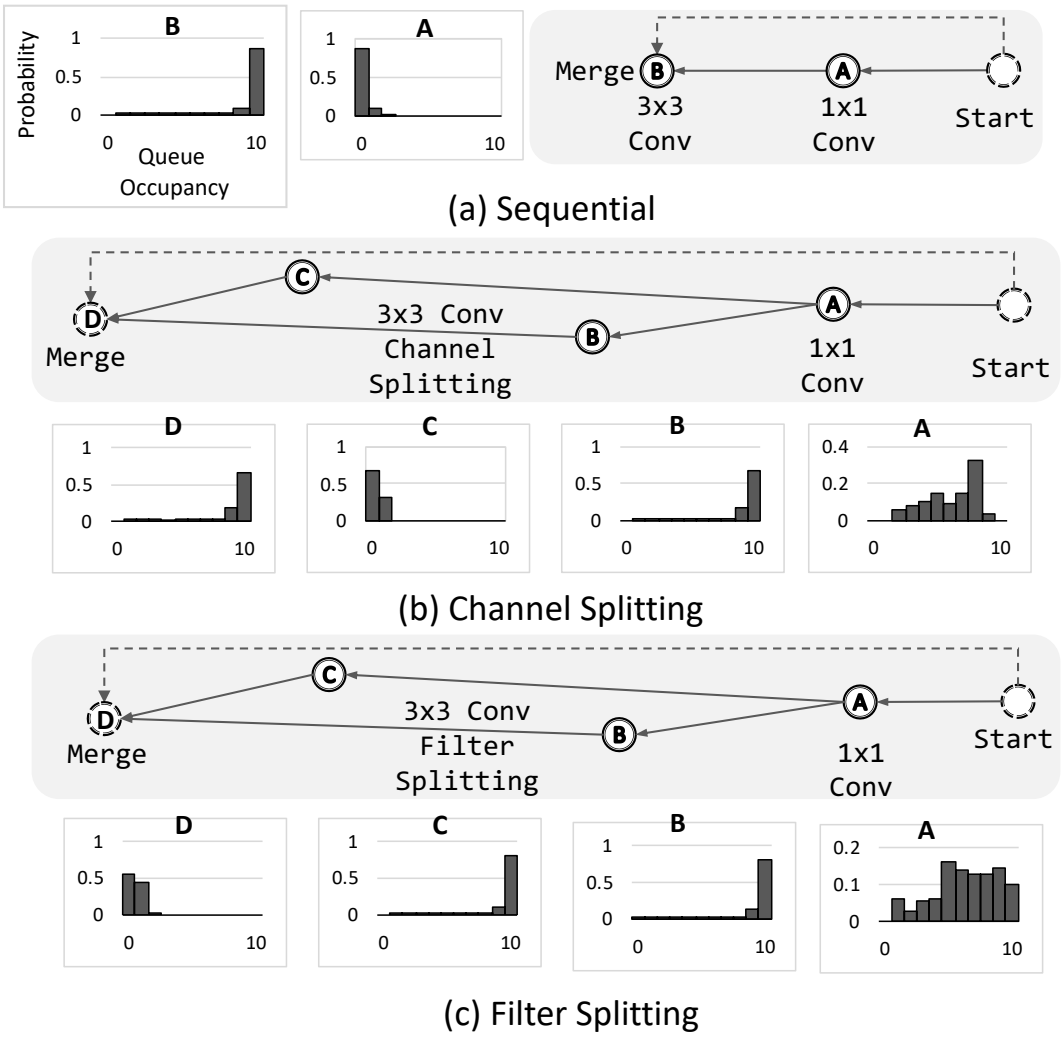


Figure 4.29: **Systems Statistics Executing Xception Block C** – See Figure 2.9. The execution of the block is shown in (a) sequential, (b) channel-splitting on two devices, and (c) filter splitting on two devices modes. The performance results are presented in Figure 4.28.

4.7 Summary

In this chapter, first, we proposed a technique to harvest the computational power of distributed robot systems by collaboration to enable efficient real-time recognition. Our technique uses model- and data-parallelism to effectively distribute computations of a DNN model among low-cost robots. We demonstrate our technique with a system consisting of Raspberry-Pi3-based robots by implementing a state-of-the art action recognition model and two well-known image recognition models. Then, we proposed several new model-parallelism methods for single-batch inferences of DNNs. We focused on DNNs for visual applications that consist mostly of CNN-based models. As discussed, with the aid of these methods, we can move the computations of DNNs closer to the edge and IoT devices. These methods divide the memory and computation footprint of DNN models and distribute them among several devices. We deployed our heuristics for several state-of-the-art visual DNN models while measuring their performance on a cluster of RPis. We have also extended this work to heterogeneous nodes by introducing IoT-tailored cluster managing tools such as Kubernetes [128]. In Chapter 6, we added reliability to such distributed systems using coded distribution [129].

Table 4.1: **Characteristics of Model Parallelism Methods for Fully-Connected Layers** – For a layer with input dimension, d_i, output dimension, d_o, and number of devices, n.

Name	**#Devices**	**Distributed Activation**	**Multiplication** (per device)	**Reduction** (per device)	**Weights** (per device)	**Communication** (total-per inference)	**Merge Operation**
No Splitting	1	N/A	$d_i d_o$	d_o	$d_i d_o$	$d_i + d_o$	N/A
Output Splitting	n	✓	$\frac{d_o}{n} d_i$	d_o/n	$n d_i$	$n d_i + d_o$	Concat
Input Splitting	n	✗	$\frac{d_i}{n} d_o$	$d_o[\lvert(d_i/n - 1)]$	$n d_o$	$d_i + n d_o$	Sum

Table 4.2: **Characteristics of Model Parallelism Methods for Convolution Layers** – Assuming same padding for a layer with input dimensions as H_ixW_ixC_i, k square filters with dimension of f.

Name	**Division Factor**	**#Nodes**	**Distributed Activation**	**Weights** (per device)	**Input** (per device)	**Filters** (per device)	**Output** (per device)	**Communication** (total-per inference)	**Merge Operation**
Baseline	N/A	1	N/A	$kC_i f^2$	$H_i W_i C_i$	$kC_i f^2$	$H_i W_i k$	$(C_i + k)(H_i W_i)$	N/A
Channel	k' $\frac{\text{filters}}{\text{node}}$	$\lceil \frac{k}{k'} \rceil$	✓	$k' C_i f^2$	$H_i W_i C_i$	$k' C_i f^2$	$H_i W_i k'$	$(\lceil \frac{k}{k'} \rceil C_i + k)(H_i W_i)$	Concat
Spatial	d $\frac{\text{part}}{\text{dimension}}$	d^2	✓	$kC_i f^2$	Eq.Equation 4.2	$kC_i f^2$	$\frac{1}{d^2} H_i W_i k$	$(d^2 Eq.Equation\ 4.2 + k)(H_i W_i)$	Concat
Filter	C_b batches	$\lceil \frac{C_i}{C_b} \rceil$	✗	$kC_b f^2$	$H_i W_i C_b$	$kC_b f^2$	$H_i W_i k$	$(C_i + k\lceil \frac{C_i}{C_b} \rceil)(H_i W_i)$	Sum

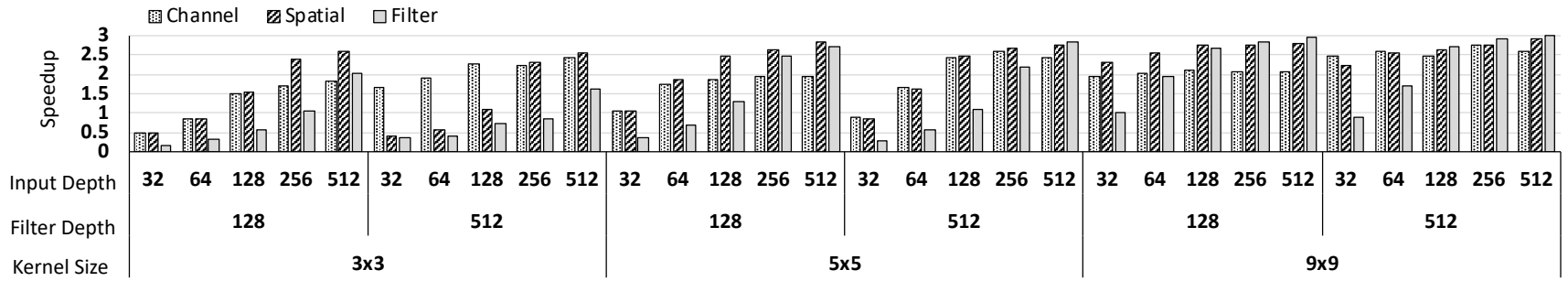


Figure 4.30: **Performance Comparison of Model-Parallelism Methods for Convolution Layers** – Convolution layers are distributed on three Raspberry Pis 3. Speedup is measured against a single Raspberry Pi.

Table 4.3: **Comparisons of Model-Parallelism Methods for Convolution Layers.**

	Channel Splitting	**Spatial Splitting**	**Filter Splitting**
Input	Entire input is copied	Input is divided spatially	Input is divided channel-wise
Filters	Some filters are saved	All filters are saved	Part of all filters are saved
Output	Each node calculates a channel	Each node calculates a spatial region	Each node calculates a partial output
Overhead	Input is copied across all devices	Input overlapping elements	Output partial sums

Table 4.4: **Raspberry Pi 3 Specifications** [41].

CPU	1.2 GHz Quad Core ARM Cortex-A53	
Memory	900 MHz 1 GB RAM LPDDR2	
GPU	No GPGPU Capability	
Price	$35 (Board) + $5 (SD Card)	
Power Consumption	Idle (No Power Gating)	1.3 W
	%100 Utilization	6.5 W
	Averaged Observed	3 W

Table 4.5: **HPC Machine Specifications.**

CPU	2x 2.00GHz 6-core Intel E5-2620	
Memory	1333 MHz 96 GB RAM DDR3	
GPU	Titan Xp (Pascal) 12 GB GDDR5X	
Total Price	$3500	
Power Consumption	Idle	125 W
	%100 Only-CPU Utilization	240 W
	%100 Only-GPU Utilization	250 W

Table 4.6: **Nvidia Jetson TX2 Specifications** [123].

CPU	2.00GHz Quad Core ARM Cortex-A57 2.00GHz Dual Denver 2	
Memory	1600 MHz 8 GB RAM LPDDR4	
GPU	Pascal Architecture - 256 CUDA Core	
Total Price	$600	
Power Consumption	Idle (Power Gated)	5 W
	%100 Utilization	15 W
	Averaged Observed	9.5 W

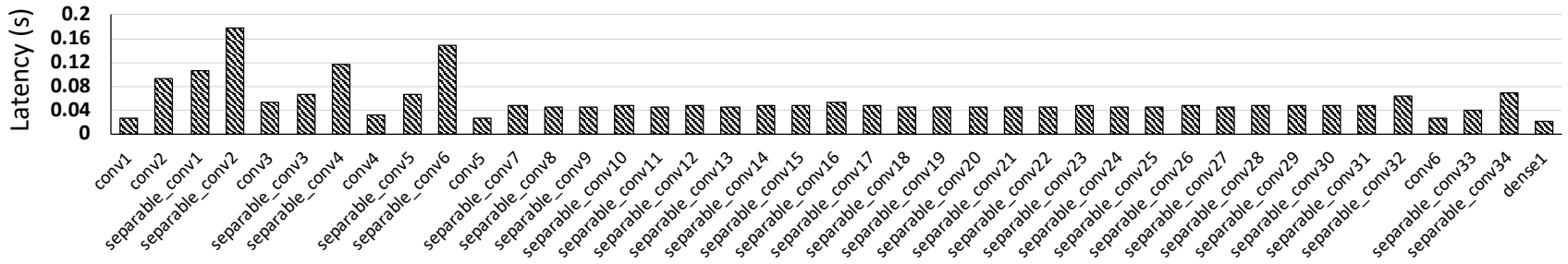


Figure 4.31: **Xception Layer-Wise Latency** – Xception measured layer-wise latency on a RPi for a single inference.

CHAPTER 5

CUSTOMIZING MODELS FOR EFFICIENT DISTRIBUTION

In this chapter, we customize DNN models (handcrafted and automatic) for more efficient distribution. This is because, with the current model architectures, both data- and model-parallelism cannot reduce communication, memory usage, and computations at the same time. Therefore, distribution incurs high communication latencies. Specifically, the communication overhead is exacerbated since edge networks usually are wireless and have unpredictable latencies (see subsection 2.3.4). In section 5.1, we first propose a handcrafted solution in customizing DNN models with low-communication parallelization (LCP) method while accompanying special hardware design [130]. In the second part of this chapter, section 5.2, we build upon the insights from the handcrafted models to propose an automatic solution to design efficient models using neural architecture search (NAS). The resulted architectures have better parallelization opportunities while providing similar accuracy performance [131].

5.1 Low-Communication Parallelization (LCP)

Our Current Approach: The current approach for enabling local DNN inference while adhering to edge devices computational and economical profile is to locally distribute inference computations by taking advantage of the existing surrounding devices such as idle IoT devices. The distribution is based on data- or model-parallelism methods proposed in the previous chapter. As an overview, in data parallelism, the entire model is duplicated on each device for performing *separate inferences*. Hence, the system needs several live and concurrent inputs to be efficient without real-time jitter. Simply put, data parallelism only increases throughput. In model parallelism, the model is divided and distributed across several devices for *the same inference* (see section 4.1 for more details).

The Key Challenge: The key challenge in above approach is that the communication overhead and the inherent inter-layer data dependency limits effective parallelism. Therefore, an ideal parallelization method for edge devices, must minimize the communication overhead, while yielding low memory and computation footprints per node. However, none of the current distribution methods jointly reduce memory usage, computations, and communication (see Table 5.1). Next subsection presents a detailed description.

Solution: To address the aforementioned challenge, we propose a low-communication parallelization (LCP) method that enables the following: ***(i)*** *Reduces Communication:* LCP models replace a single, wide, and deep model with several narrow ones that only communicate for input and pre-final activations. Thus, their communication load is low with distributions (see Table 5.1). ***(ii)*** *Reduces Compute and Memory Footprints Per Node:* LCP models have fewer connections than those of the original ones, so their number of parameters and computational demands are also lower than those of the peer model-parallelism versions, shown in Table 5.1. ***(iii)*** *Allows Inter-Layer Parallelism:* Narrow branches in LCP models are independent of each other, which enables inter-layer parallelism. This is in contrast to model parallelism that only allows intra-layer parallelism due to the single-chain dependency between consecutive layers. ***(iv)****: Recovers Accuracy with no Additional Parameters:* After splitting the model into branches, to recover a possible accuracy loss, LCP may slightly fatten the branches. However, since it reduces unnecessary communications, the overall parameters after fattening are still fewer than the original one.

LCP is orthogonal and an addition to current techniques such as weight pruning [15] and quantization [62] that reduce the computational demand of models. LCP models offer distribution/parallelism opportunities for distributed computing, whereas current techniques apply accuracy/performance tradeoffs to single-node models. Such techniques can be applied to each branch of our method, as shown in subsection 5.1.4. Thus, LCP complement such techniques rather than compete with them.

Experiments Overview: **(1)** We generate and evaluate LCP models based on image-

Table 5.1: **Distributing Methods Overview:** Comparison of distribution methods for inference.

	Data Parallelism	**Model Parallelism**	**Target**	**LCP**
Memory Per Device	DNN	$\frac{1}{n}$DNN	$\frac{1}{n}$DNN	$\leq \frac{1}{n}$DNN
Communication Per Inference	IN/OUT	Intermediates +IN/OUT	IN/OUT	$\approx$ IN/OUT
Computation Per Device	DNN	$\frac{1}{n}$DNN	$\frac{1}{n}$DNN	$\leq \frac{1}{n}$DNN

DNN: Metrics associated with the entire model; n: Number of devices.

recognition DNNs on MNIST [43], CIFAR10/100 [44], Flower102 [45], and ImageNet [46] datasets (total of 53 training results), covering all MLPerf [48] image-recognition models. **(2)** To evaluate the execution performance of our method, we conduct real-world implementations on three distributed systems with up to 10 Raspberry Pis (RPis), two PYNQ boards, and up to eight AWS instances. RPis are chosen because they represent the de facto choice for several robotic and edge use cases and they are readily available [132, 133, 134, 135, 136]. **(3)** We also evaluate the performance of LCP on customized hardware. Because, besides tailoring models based on hardware limitations, the architecture of hardware could be tailored to better achieve the goal of fast inference. To this end, we slightly modify the architecture of TPU [47] to make it latency-optimized for edge applications rather than throughput-optimized, and implement it on a small Xilinx FPGA. **(4)** To further investigate area and power efficiency of our tailored hardware for integrating with edge devices, we implement an ASIC chip in ASAP 7 nm [137].

Contributions: The contribution of this section are as follows:

- We propose the first DNN parallelization method to reduce the communication overhead for distributed inference.

  We generate LCP models, with inter-layer parallelism for fast inference at small memory and computation footprints.

- We investigate the impact of hardware/software co-design on inference performance, by tailoring the hardware of TPU [47] for optimizing single-batch inference latency, and implement it on a small FPGA and as a tiny 0.107mm^2 low-power chip consuming only 16mW.

5.1.1 Challenges of Previous Distribution Methods

Besides the challenges of growing DNNs, single device pareto frontiers and the high communication overhead that are introduced in section 2.3, the studied distribution methods in previous chapter also have limitations.

As we know, first Data parallelism (Figure 5.1a) parallelizes the computations of independent inputs [1, 65]. Therefore, data parallelism does not apply to the edge because: It (i) serves several independent requests, the number of which is limited in an edge environment; (ii) does not reduce *end-to-end latency* per inference and only increases throughput. Latency is important in several applications in the edge; and (iii) does not change the computation and memory footprints per node (Table 5.1).

Furthermore, as discussed in Chapter 4, Model-parallelism (Figure 5.1b) divides the inference computations for the same request [1, 65]. This method divides the computations within layer(s) while keeping dependencies intact. Depending on the type of layer, the dividing could take several forms. For a simple example for distributing a fully connected (`fc`) layer, there are two extremes of model parallelism: Input and output splitting [116] (Chapter 4). In output splitting, producing each output(s) is divided among the devices. In input splitting, the input is split and each device computes all parts of the output that are dependent on their received input. Each method has communication overhead (transmission of the input to all nodes or partial sums to a final node for summation). We crafted New model-parallelism methods by mixing these two extremes, but they similarly suffer from the same discussed overhead. This is because model parallelism does not change the interconnection of a model. Hence, although model parallelism reduces the compute and

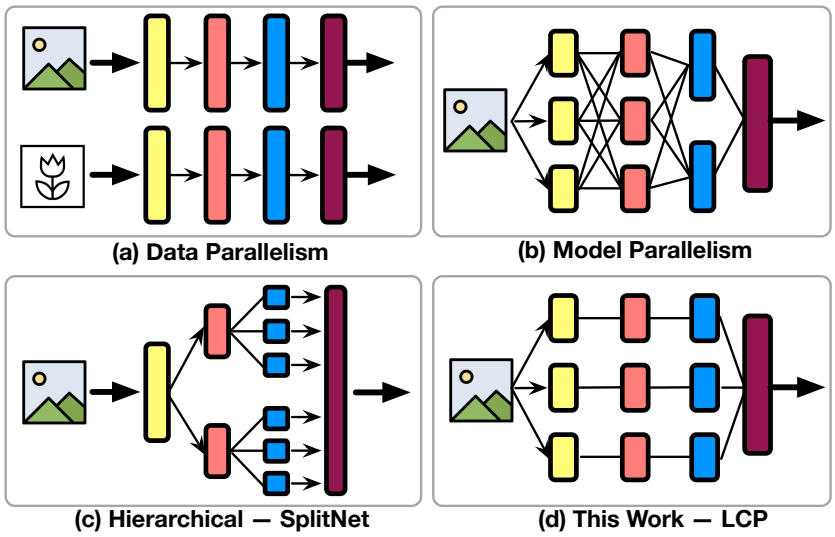


Figure 5.1: **Overview Distribution Methods:** (a) data parallelism, (b) model parallelism. (c) hierarchical – SplitNet [80], and (d) LCP

memory footprint per node; the single-chain dependency between consecutive layers limits the parallelism scope within a single inference and causes communication overhead.

To solve this challenge, SplitNet [80], shown in Figure 5.1c, gradually splits the model in a tree-structured style *manually* based on the dataset semantics, extracted in intermediate to final layers. Therefore, SplitNet (i) splits only intermediate to final layers, (ii) is invariant to the number devices, (iii) creates imbalanced workload because of its dependency on semantics, (iv) results in tree-style connections, incurring high communication overhead, and (v) enforces a new splitting when dataset changes.

Therefore, an ideal distribution method for edge devices besides yielding low memory and computation footprints per node must consider communication overhead. The single-chain dependency between consecutive layers limits the available parallelism that could be harvested by the aforementioned methods. The limitation is that after the computations of a single/few layer(s) are done, the intermediate results must be merged before being forwarded to the next layer. Such merging acts as a global barrier, which similar to parallel programming, limits the gained performance speedup. In summary, with parallel execution

on multiple devices, ideally, we could pass the frontier in Figure 5.2. However current distribution methods are limited by the communication overhead and the inherent inter-layer data dependency. The next section proposes LCP models, which significantly reduce communication and allow inter-layer parallelism.

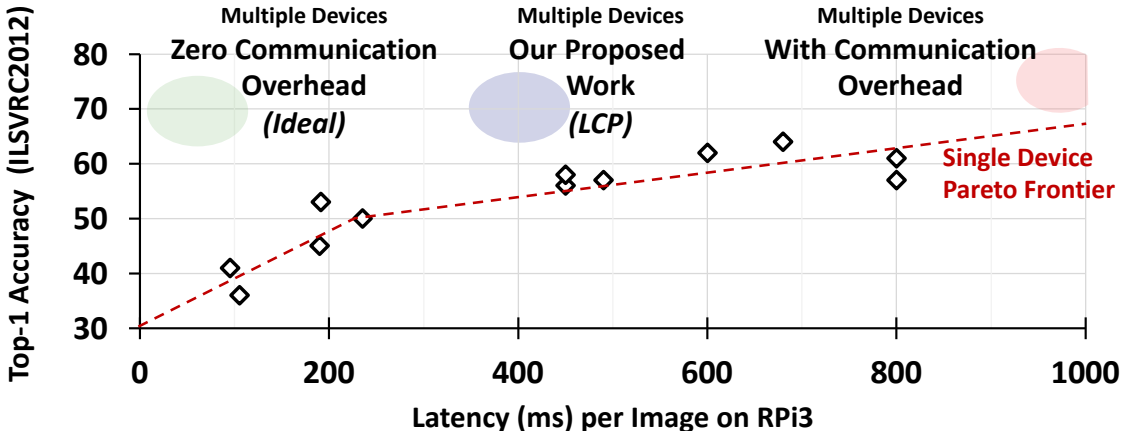


Figure 5.2: **Latency-Accuracy Pareto Frontier – Single device:** Latency per image on RPi3 for ILSVRC models with the optimized platform-specific compilation ELL [29] tool [61]. Multiple devices: Breaking the single device Pareto frontier, but with significant communication overhead.

5.1.2 LCP for Fast Inference

To address challenges, we propose LCP method, which replaces a single, wide, and deep model with several narrow branches that only communicate for input and pre-final activation (Figure 5.1d). Figure 5.3b shows an example of a two-branch LCP model for VGG-S. This subsection explains the design procedure of LCP models and discusses their key features enabling low-communication parallelization.

LCP Design Procedure

Figure 5.4 describes the design procedure of LCP models. We start by inputting the DNN model and its per-layer memory and computation footprints. Similarly, we input the specification of the hardware, such as memory size, computation capability, and any

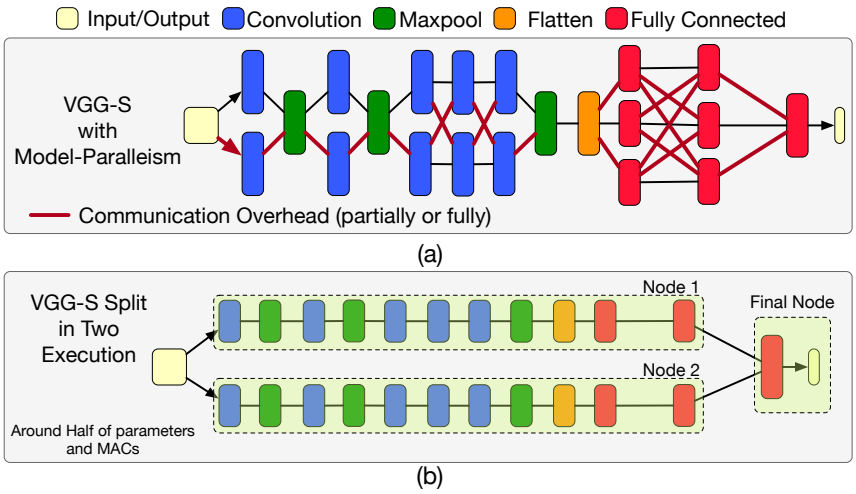


Figure 5.3: **VGG-S Distributions:** (a) model parallelism and (b) LCP versions.

overhead associated with executing a DNN on our hardware. For instance, several DNN frameworks have a memory overhead because of the framework. A splitter procedure, described in Algorithm 3, in a while loop, splits the model, cuts the connection, and measures the approximate footprints of each branch. The $\text{Division}_{\text{Factor}}$, a hyperparameter, defines the granularity of division/splitting. Here, we assume the $\text{Division}_{\text{Factor}}$ of two, but any number is viable. The loop exits when a single branch is fitted on a device (both memory and computation wise). If the number of devices is fewer than the number of branches, the execution is still possible, but will be inefficient. Then, we remove non-branch connections in a simple operation that keeps only one connection per layer. Figure 5.3b illustrates an LCP model with two branches.

The derived model from the splitter is the *split-only model*. By training the split-only model and testing it, we measure its accuracy. The split-only models have fewer parameters and MAC operations than the original models (see Table 5.2) in total. Hence, after distribution, each branch has less computation and memory footprint than its model-parallelism version.

As a result of fewer number of parameters and removing several connections, a slight accuracy drop in split-only LCP models is expected. Depending on the accuracy requirement of the task, we either fatten each branch by $F\%$, a hyperparameter, or output the resulted

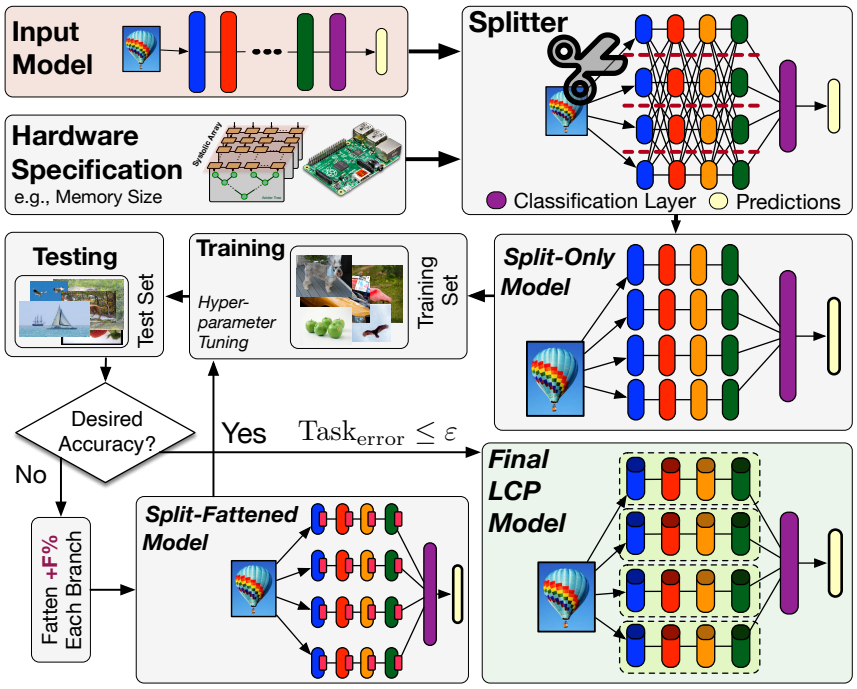


Figure 5.4: **LCP Design Procedure:** Overview of steps in designing LCP models. Splitter details are presented in Algorithm 3.

model. We assumed a maximum of 3% bound for $\text{Task}_{\text{error}}$. Fattening each branch by $F\%$ is done by increasing the number of channels and output features of convolution and fully connected layers of the split-only model, respectively. Note that theses new *split-fattened models* are fattened within each branch. Thus, even with a high fattening percentage, still they have fewer parameters and MAC operations than the original model (see Table 5.3). When the accuracy is in the acceptable error range for our task, $\text{Task}_{\text{error}}$, we output the model architecture and its weights. It is expected that with similar number of parameters after fattening, LCP models achieve the same level of accuracy [105]. We showcase LCP models in subsection 5.1.4 covering MLPerf [48].

Algorithm 3 LCP Splitter (in Figure 5.4)

Inputs: DNN: Layer configurations $[0 : n]$
DNN_{Mem}, DNN_{MAC}: DNN memory and computational footprints
$\text{Division}_{\text{factor}}$: Division Factor for splitting
Dev_{Mem}, Dev_{MAC}: Hardware specification
Output: DNN Layer configurations $[1 : n]$
function SPLIT(DNN, DNN_{Mem}, DNN_{MAC}, $\text{Division}_{\text{factor}}$, Dev_{Mem}, Dev_{MAC})

```
    mem ← 0
    mac ← 0
    while not mem and not mac do
        mem ← DNN_Mem < Dev_Mem
        mac ← DNN_MAC < Dev_MAC
        for layer [0..n − 1] in DNN do
            layer.width ← layer.width / Division_factor
        DNN ← RemoveNonBranchConnection(DNN)
    return DNN
```

Key Features of LCP Models: LCP models are designed by considering their underlying computation domain and have the following key features to address the challenges discussed in section 2.3 and additionally in subsection 5.1.1: **(1)** LCP models only communicate for input and pre-final activation. Therefore, they significantly reduce communication overhead in a distributed system. Additionally, the low communication load per inference helps with the straggler problem. This is in contrast to model parallelism, which highly depends on communication among all the intermediate layers; **(2)** LCP models split the size of a layer, so the total parameter size and computation complexity of the model are reduced. Therefore, they require fewer parameter sizes, less computation complexity, and no communication between the nodes for intermediate layers. These lower memory and computation footprints allow edge devices to efficiently operate within their limited resources (*e.g.*, no swap space activities due to limited memory); **(3)** LCP models replace the original wide model with several narrow and independent branches. Since the computations of branches are not dependent, in contrast to the single-chain of dependency in the original model, the scope of parallelism is not limited with each layer anymore. In other words, LCP models go beyond intra-layer parallelism.

5.1.3 LCP Hardware Design

Last section showed how we enable fast inference under resource constraints and at costly communication, by proposing a low-communication parallelization method that results in several narrow models. To further achieve the goal of fast inference and recognize the potential, the hardware can also be tailored. Recently, several popular tailored hardware designs for DNNs [86, 87, 88, 47, 85] including TPU [47] use systolic arrays [138] that offer a high degree of concurrent processing through a dataflow compute arrays hence providing high *throughput*. In the edge applications, however, the main goal is *reducing single-batch inference latency*, rather than high throughput solely. This section introduces our microarchitecture (Figure 5.5a), an example of tailoring and simplifying the architecture of TPU to be implemented on small FPGAs or be fabricated as tiny (*i.e.*, 0.107 mm^2 as shown in Figure 5.5b, more details in subsubsection 5.1.4) low-power chips to be integrated with edge devices.

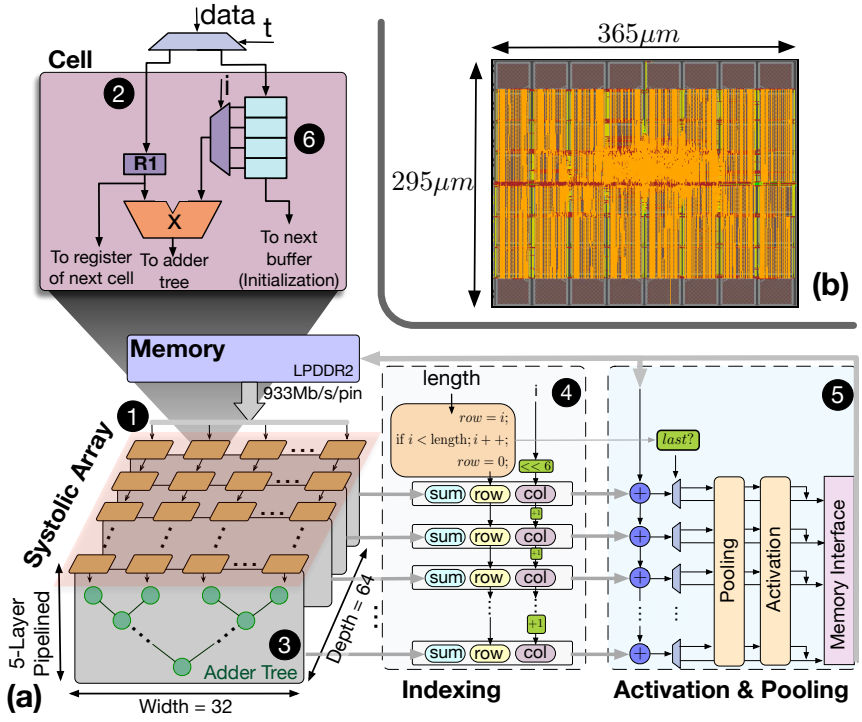


Figure 5.5: **Details of Tailored Hardware for Edge:** (a) Microarchitecture overview, and (b) Layout of ASIC design at 7nm.

Figure 5.5a illustrates our tailored microarchitecture that similar to TPU, comprises a weight-stationary systolic array [138] for implementing matrix-matrix multiplication. The systolic array cells are organized in a 32x64 array ❶. To reduce the number of connections, only the first row of the systolic array is connected to the memory ❶. Moreover, each cell of the first row is only connected to one data stream line ❷. Based on the type of an operand (t), streaming data is used for either initialization or for processing. Since the width of the systolic array is 32, a heuristic algorithm partitions the streaming (i.e., non-stationary operand) into blocks of 32 width and arbitrary length, and splits the stationary operand, into 32×64 blocks. To assist the smooth streaming of data from memory to the systolic array, we map these blocks along with their indices (i), type (stationary/non-stationary), and length to sequential memory addresses. We implement our 32×64 systolic array connected to *LPDDR2* memory with the data rate of 933Mb/s/pin @466 MHz [139], which gives a bandwidth of 3.7 GB/s. Other packaging options with higher memory bandwidths are also feasible. However, seeking a fair comparison with RPi3s, we choose this memory technology. The maximum data reuse rate of our design is 64 OPs/Byte, which leads to a peak throughput of 217.6 GOPs/s. The following explains three main modifications we made to this systolic architecture, to achieve our goal of *reducing single-batch latency*.

(1) Adder Trees: Instead of MAC-based systolic arrays, we separate adders from multiplications by integrating adder trees, the well established components for DNN accelerators [92, 140, 141], into systolic arrays architecture. Each cell of our systolic array is a *multiplier* with two integer operands, one stationary and one streaming (R1). Each row of the multiplier array is connected to an adder tree ❸, pipelined in five ($log_2 32$) stages. Adder trees reduce the result of multiplications into a single integer, which then contributes to creating an output element. The structure of the multiplier array connected to the adder trees reduces latency from O(n) to O(log(n)).

(2) Simple Indexing Logic: We use a data-driven execution model, in which data is pushed by the memory to the systolic array, triggered by the arrival of data. During execution, for

each element, the indexing logic (❹) generates the appropriate row and column indices of the element using the index (i) of a block and its length to accompany the result. The row and column indices will later be used by the memory interface to write the results to physical locations in memory. By comparing the length and index (i), the end of the operations in the current layer is detected. The end of the current layer signals the start of activation and pooling functions (❺) for that layer.

(3) Buffering Stationary Operands: The stationary operands are often larger than the depth of the systolic array. In such cases, we have to partition a multiplication into several small operations that share a non-stationary operand, but have distinct stationary operands. To avoid multiple loads of stationary registers, we choose to integrate a buffer (❻) for stationary operands at each cell. As a result, the design serves requests with lower latency. Moreover, since each branch of the model has several layers, integrating these buffers allows fast context switching without the overhead of reloading the stationary operands. These buffers are connected in a column of cells, similar to streaming registers (`R1`)s. During the initialization, stationary operands are poured into these connected buffers to fill them by utilizing the connections between them.

5.1.4 LCP Experimental Studies

This section shares our experimental results for generating LCP models and then their full-system implementation on RPi, TVM [84] on PYNQ boards, and AWS servers. Finally, we evaluated our hardware with edge FPGA implementation, and ASIC chip design. At the start of each subsection, the setup of related experiments is provided.

Generating LCP Models

Training Specifications: We train all the models, including the original model, from scratch to conduct a fair comparison. Normalization [142] layers are included. The training is done with an exponential learning rate with a decay factor of 0.94, initial learning rate 1e−2,

number of epoch per decay of two or 10, a dropout rate of 50%, and L2 regularization with weight decay of 5e−4. We use ADAM optimizer [143] with $\beta_1 = 0.9$ and $\beta_2 = 0.99$. All biases are initialized to zeros and all weights are initialized with a normal distribution of mean 0 and a standard deviation of 4e−2. All of our models are trained until the loss is flattened or least for 12 epochs. Test and accuracy measurements are done on at least 10% of datasets that have never been used in training to provide an unbiased evaluation of the model. For LCP, the $\text{Division}_{\text{Factor}}$, F, and ε, are 2%, 10%, and $\approx 3\%$, respectively.

Datasets: We use the following datasets: (1) MNIST [43], which contains 70k grayscale handwritten 28x28 images in 10 classes; (2) CIFAR10 [44], which contains 60k colored 32x32 images in 10 classes; (3) CIFAR100 [44], which contains 60k colored 32x32 images in 100 classes; (4) Flower102 [45], which contains 16,378 colored 224x224 images of flowers in 102 classes; and (5) ImageNet [46], which contains 1.33 M colored 224x224 images in 1000 classes.

Models: We use the representative model for each dataset, LeNet [54], LeNet-FC [54], VGG-S [37], CifarNet [44], VGG16 [37], AlexNetv2 [66], ResNet-18/50 [36], and MobileNet [144]. We cover all image-recognition models in MLPerf. In total, for brevity, we only report 53 instances of training results to show LCP extensibility using five datasets and nine models. Our additional results (not reported) with ResNet-34, DenseNet [78], and DarkNet19 [145] confirms extendibility. Simple sequential DNNs serve as a basis to confirm our method, while ResNets and MobileNet showcase LCP with modern models.

Split-Only Models: For split-only models, we use $\text{Division}_{\text{Factor}}$ of two, which results in models with two, four, and eight branches. Except the width, defined as output features in fully connected layers and the number of output channels (i.e., filters) in convolution layers, the rest of the parameters are similar to the original model as Splitter Procedure Algorithm 3 only touches widths. Table 5.2 lists the training results. Figure 5.6a illustrates the accuracy difference of our models, shown in Table 5.2. As shown, the maximum accuracy drop is around 5% for CifarNet. Note that this accuracy drop occurs when we reduced the parameter

Table 5.2: **Split-Only LCP Models Training:** Results of split-only LCP models.

Model Name	Dataset	Layers†	Top-1 Accuracy	# Param	# MAC Opr.
LeNet-FC*	**MNIST**	**3fc**	**97.95**	**266.6k**	**266.2k**
LeNet	**MNIST**	**2fc-3c-2p**	**98.76**	**61.7k**	**61.5k**
LeNet-split2	MNIST	3fc-6c-4p	98.86	31.5k	30.5k
LeNet-split4	MNIST	5fc-12c-8p	98.93	16.1k	16.0k
LeNet-split8	MNIST	9fc-24c-16p	98.81	8.8k	8.5k
CifarNet*	**Cifar10**	**2fc-2c-2p-2n-1d**	**80.72**	**797.97k**	**14.79M**
CifarNet-split2	Cifar10	5fc-4c-4p-4n-2d	80.63	401.75k	9.32M
CifarNet-split4	Cifar10	9fc-8c-8p-8n-4d	79.53	203.64k	6.59M
CifarNet-split8	Cifar10	17fc-16c-16p-16n-8d	76.70	104.6k	5.22M
CifarNet	**Cifar100**	**2fc-2c-2p-2n-1d**	**52.87**	**815.34k**	**14.81M**
CifarNet-split2	Cifar100	5fc-4c-4p-4n-2d	51.22	410.48k	9.33M
CifarNet-split4	Cifar100	9fc-8c-8p-8n-4d	48.48	208.05k	6.59M
CifarNet-split8	Cifar100	17fc-16c-16p-16n-8d	47.98	106.85k	5.23M
VGG-S*	**Cifar100**	**3fc-5c-2p-1n-2d**	**50.33**	**76.15M**	**154.09M**
VGG-S-split2	Cifar100	5fc-10c-4p-2n-4d	48.53	38.09M	78.28M
VGG-S-split4	Cifar100	9fc-20c-8p-4n-8d	47.72	19.06M	40.37M
VGG-S-split8	Cifar100	17fc-40c-16p-8n-16d	48.48	9.54M	21.42M
VGG-S	**Flower102**	**3fc-5c-3p-1n-2d**	**88.14**	**60.79M**	**1.85G**
VGG-S-split2	Flower102	5fc-10c-6p-2n-4d	89.31	30.50M	1.01G
VGG-S-split4	Flower102	9fc-20c-12p-4n-8d	87.55	15.26M	591.65M
VGG-S-split8	Flower102	17fc-40c-24p-8n-16d	85.66	7.64M	382.51M
ResNet-18	**ImageNet**	**18c-2p-17n**	**70.68**	**11.69M**	**1.82G**
ResNet-18-split2	ImageNet	35c-3p-34n	69.85	6.11M	0.98G
ResNet-18-split4	ImageNet	69c-5p-68n	68.07	3.32M	0.55G
ResNet-18-split8	ImageNet	137c-9p-136n	66.76	1.93M	0.34G

† fc: fully-connected, c: convolution, p: pooling, n: normalization, and d: dropout.
* Detailed results are removed for brevity, refer to Figure 5.6. The results follows the same trend.

size of our model extensively (around 1/8). Figure 5.6b and c show reduction in the number of parameters and computation compared with the original DNN model; as seen, each split reduces both by about split_{factor} times. This is because each convolution and fully connected layer in the split version create fewer outputs; therefore, the next layer requires fewer parameters. In the next section, we restore the accuracy of LCP models with split-fattened models.

Split-Fattened Models Accuracy is a defining factor in several applications. Thus, we provide a remedy to restore the accuracy of split-only models. By fattening (*i.e.*, adding more parameters) each branch, we aim to create larger layers in the split-only models. To do so, for each layer (excluding classification layer) in every branch, we increase the width by a fraction. So, fattening by 20% means the size of the output in each layer is increased 1.2x. We fatten every branch in 10% steps as Algorithm 3 shows. Our experiments focus on split8, which have the highest accuracy drops. Figure 5.7 shows a summary of these models. As seen, 40% split-fattened models have higher accuracy than the original model while

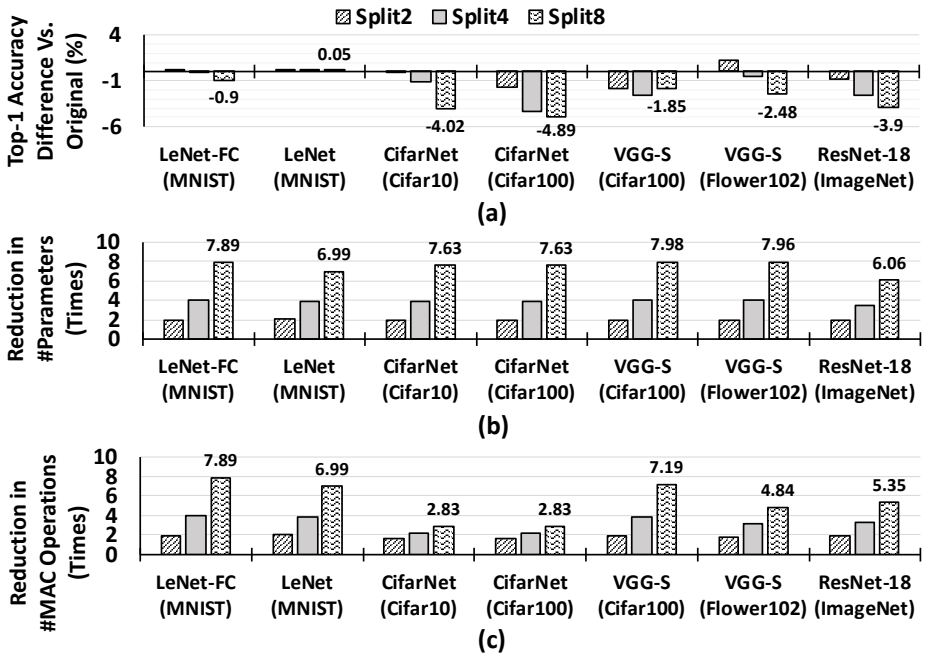


Figure 5.6: **Split-Only Models:** (a) Accuracy, (b) reduction in the number of parameters, and (c) reduction in the number of MAC operations in comparison with the original model.

having fewer parameters and MAC operations. On average (for 30% and 40% models), with 4.61x–3.81x fewer parameters and 2.95x–2.5x fewer MAC operations, split-fattened models achieve accuracy within our error bound of 3%, $\text{Task}_{\text{error}}$, while they jointly optimize memory, computation, and communication for edge.

ImageNet Models: Table 5.3 illustrates the results of ImageNet models. For the sake of brevity, we only show split8 and one fattened model. As shown, `f40` models restore the accuracy within 3% of the original model. The tradeoff for 3% accuracy loss is about 4x fewer parameters, 4x fewer computations, and 8x less communication load (vs. model parallelism). Figure 5.8 presents a comparative analysis for the communication load between distributed original models with model parallelism and distributed LCP models. Since LCP models avoid communication between their branches, the communication load is reduced significantly. In short, although split models are more complex than the original models in terms of the number of layers and connections, they achieve more parallelism with less communication load.

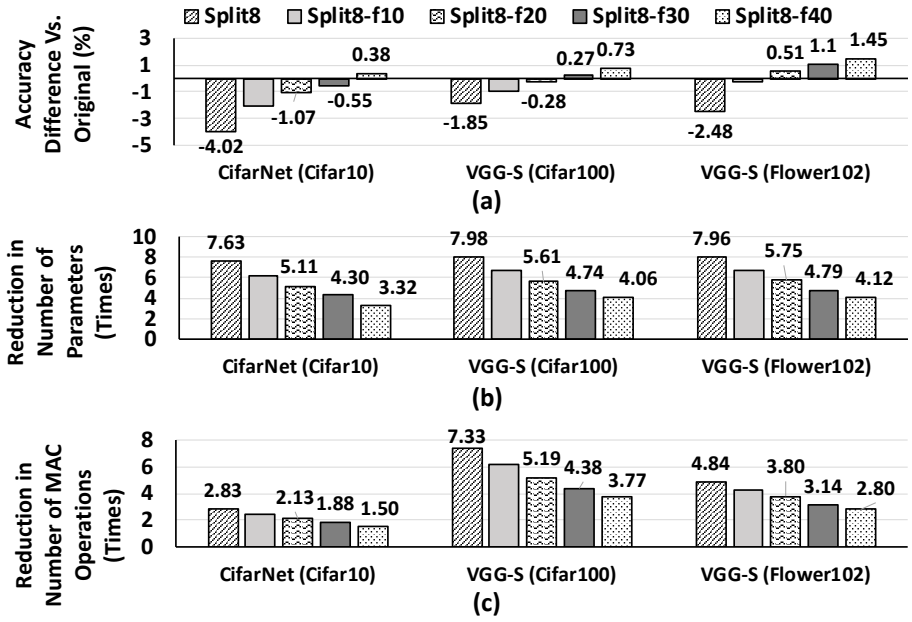


Figure 5.7: **Split-Fattened Models** – Common visual models (a) Accuracy difference, (b) reduction in the number of parameters, and (c) reduction in the number of MAC operations in comparison with the original one (Table 5.2).

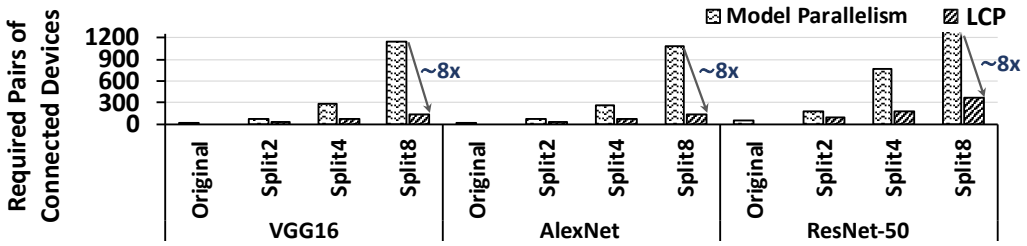


Figure 5.8: **Reduction of Communication with LCP**: Communication reduction with LCP models compared to model parallelism (required pairs of connections).

Exploring Performance on RPis, PYNQs, and AWS

RPi Experiments Setup: To study the benefits of LCP models versus only model-parallelism methods, we deploy several models on a distributed system of Raspberry Pi 3s (RPis), the specifications in Table 5.4. On each RPi, with the Ubuntu 16.04 operating system, we use TensorFlow [117] and Apache Avro [125], a remote procedure call (RPC) and data serialization framework, for communication between RPis. We measure power using a USB digital multimeter [146]. A local WiFi network with the measured bandwidth of 62.24 Mbps and a measured client-to-client latency of 8.83 ms for 64 B is used. All the

Table 5.3: **ImageNet LCP Models Training:** Results of ImageNet LCP models.

Model Name	Dataset	Top-1 Acc.	Top-5 Acc.	# Param.	# MAC MAC Opr.
AlexNet	**ImageNet**	**57.02**	**80.32**	**50.3M**	**678.97M**
AlexNet-split8	ImageNet	49.03	73.10	6.32M	145.37M
AlexNet-split8-f40	ImageNet	54.68	77.06	12.11M	244M
VGG16	**ImageNet**	**70.48**	**90.02**	**138.36M**	**15.47G**
VGG16-split8	ImageNet	58.67	81.54	7.64M	2.01G
VGG16-split8-f40	ImageNet	67.24	89.23	33.78M	3.87G
ResNet-50	**ImageNet**	**75.4**	**93.1**	**22.80M**	**4.87G**
ResNet-split8	ImageNet	61.79	81.22	5.42M	0.88G
ResNet-split8-f40	ImageNet	72.12	92.19	8.60M	1.18G
MobileNet	**ImageNet**	**71.7**	**90**	**4.24M**	**4.86G**
MobileNet-split8	ImageNet	59.68	83.23	1.12M	0.93G
MobileNet-split8-f40	ImageNet	68.05	89.12	2.12M	1.34G

For `[model_name]-f[number]`, number represent the percentage of fattening.

real-world experiments are full-system measurements with all overheads included without any simulations/estimations.

RPi Performance & Energy: Figure 5.27 presents latency of inference per image on RPis. On a single device, AlexNet has 2.8 seconds latency, while VGG16 achieves 9.4 seconds latency. By deploying model-parallelism variants of the models on four and eight RPis, we achieve a maximum of 0.42s latency, a 6.6x increase, for AlexNet. But, for VGG16, on four RPis, we observe a slowdown, which is caused by high communication latency. LCP variants of split4 and split8 can reach up to 115 ms and 400 ms latency per image for AlexNet and VGG16, respectively. This is because LCP models are lightweight and parallelizable and have low communication. Figure 5.9 shows measured energy per inference for RPi implementations. To compare with previous related work, SplitNet [80], Figure 5.27

Table 5.4: **Platform Specifications**: Specification of RPi, PYNQ FPGA, and AWS.

Raspberry Pi 3B+

CPU	1.2 GHz Quad Core ARM Cortex-A53
Memory	1 GB LPDDR2 SDRAM @ **933Mb/s/pin**
Die Size	$\approx 196mm^2$ @ **28 nm**

Edge FPGA (Zynq Artix 7 XC7Z020)

		DSP48E	FF	LUT
Utilization	#Unit	96	5427	2343
	%	44	5	4
Static Power	0.121 W			
Dynamic Power	Signals: 0.009 W Logic: 0.003 W			

AWS

AWS Instance	T2.micro
Specification	1 vCPU, 1 GB Memory, 64 GB Storage

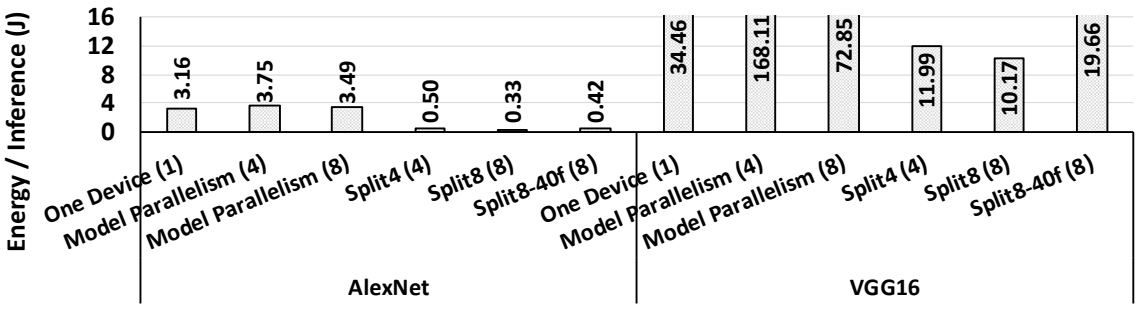


Figure 5.9: **Devices Total Energy per Inference:** Model-parallelism, and LCP on RPi (number in parenthesis is #devices) total energy per inference.

presents the performance of SplitNet models for AlexNet with different configurations. As seen, the performance is worse than LCP models. This is because SplitNet creates more merging/synchronization points with its tree-structured model design. The resulting model exponentially introduces more merging/synchronization with increased depth, which also does not equally split all the layers (causing load balancing issues). Finally, SplitNet performs parallelization based on dataset semantics, which means every dataset and model needs to be manually split.

TVM Experiments on PYNQ Boards: As a real-world example for edge FPGA implementation, we use TVM [84] on the PYNQ [147] board. PYNQ is designed for embedded applications. We use the TVM VTA stack on the PYNQ as the architecture (RISC-style instructions) and only change the models (ResNet-18 vs. LCP ResNet-18 Split2 with <1 accuracy drop). In this way, we can measure the benefits of LCP models without relying on any special tailored hardware. Our performance result shares the entire system pipeline

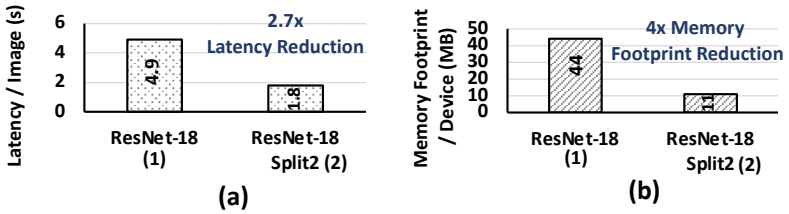


Figure 5.10: **TVM Experiments:** (a) Latency per image, (b) memory footprint per device (number in parenthesis is #devices).

performance, from a live camera feed to prediction output on two boards versus one board. Figure 5.10a shows a 2.7x speedup, including all communication and system overheads, network latency, and jitter because LCP models are parallelized on two devices and, in total, they have lower computation and memory footprints. The measured reduction in memory footprint is shown Figure 5.10b.

AWS Experiments: To see the reduced communication and distributed execution benefits of LCP models further, we deploy AlexNet, VGG16, and ResNet-50 models on AWS T2.micro instances with only one vCPU and 1 GB memory per instance. Figure 5.11 presents the derived statistics. In all cases, LCP models not only reduces the average latency but also significantly reduce maximum latency. Splits four and eight have lower speedup compared with our RPi experiments because all the 4/8 instances are not hosted on the same machine; thus, the communication cost is higher than the usual edge-specific cases that this paper targets.

Edge FPGA Experiments

FPGA Experiments Setup: We implement our tailored microarchitecture on a ZYNQ XC7Z020 FPGA targeting PYNQ-z1 boards. We use Xilinx Vivado HLS for implementation and verify the functionality of our implementation using regression tests. We use relevant *#pragrma* as hints to describe our desired microarchitectures in C++. We synthesize and

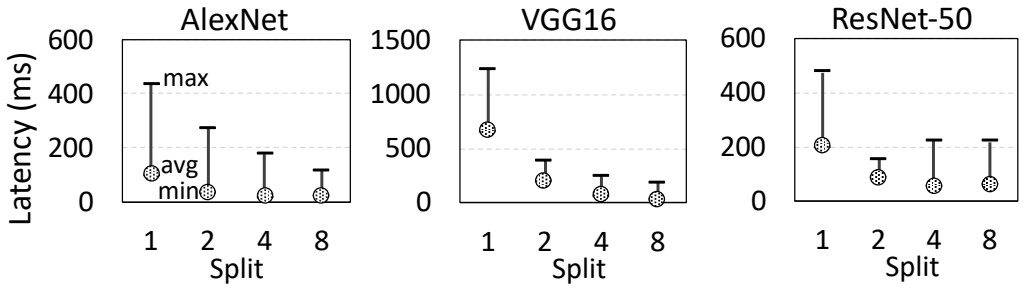


Figure 5.11: **Amazon AWS Experiments**: Average, minimum, and maximum latencies of distributed LCP execution on AWS T2.micro instances with 1 vCPU and 1 GB memory per instance.

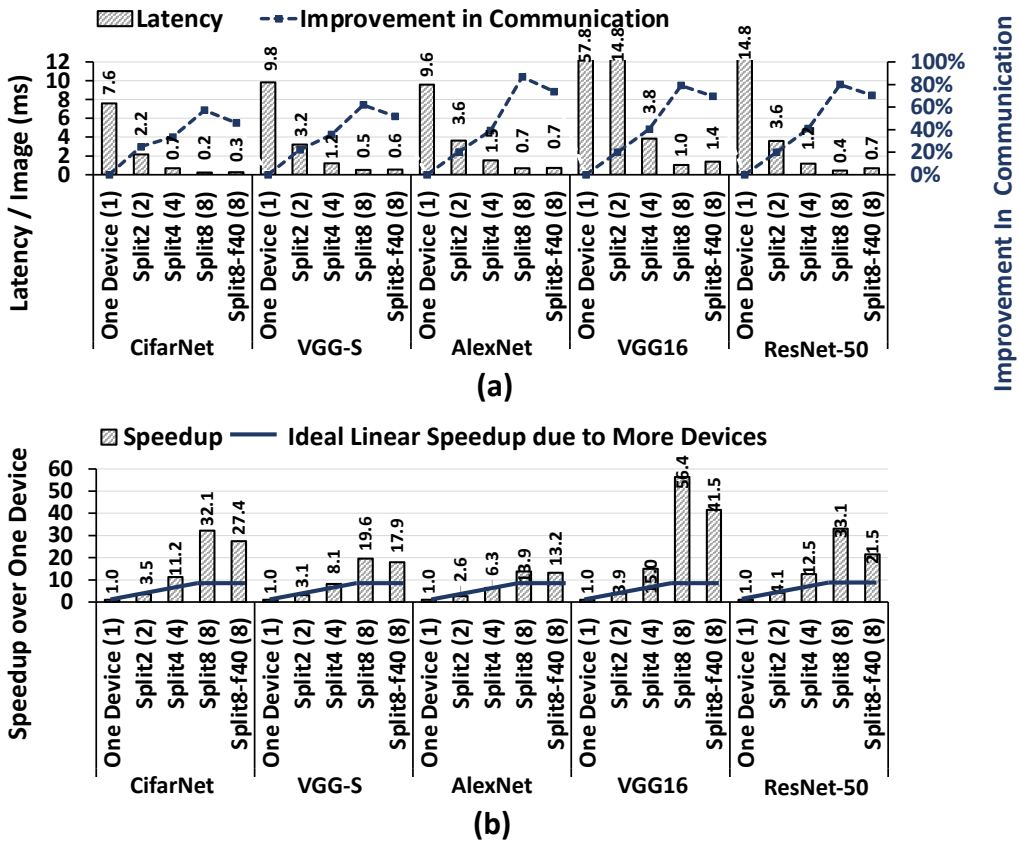


Figure 5.12: **Edge FPGA with Tailored Hardware Latency and Speedup:** (a) Latency per image, (b) speedup over one device (number in parenthesis is #devices).

implement our design using Vivado and report post-implementation (*i.e.*, place & route) performance numbers and resource utilizations. Inputs and output of our design are transferred through the AXI stream interface. The clock frequency is set to 100 MHz. Communication for multiple devices is estimated with the network provided in subsubsection 5.1.4.

FPGA Performance: Figure 5.12 shows the experiment results for our edge-tailored hardware. The latency per image is shown in Figure 5.12a, with improvement in communication overhead versus model-parallelism methods (86% and 60% for 8split and 4split). Depending on the model, the inference per latency on a single device is between 4–29ms; a 221–325x speedup compared to RPi results for AlexNet and VGG16. Our designed LCP models achieve acceptable performance for edge computing, which is 10s of inferences per second, around 1–10ms. As observed, the accuracy loss of our split-only models can be easily

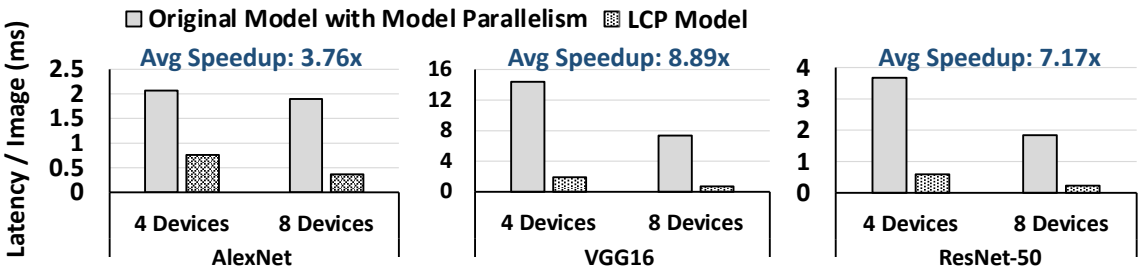


Figure 5.13: **Latency per Image for Edge FPGA with Tailored Hardware:** Comparing LCP versus model parallelism.

restored by fast split-fattened models of `f40` with a negligible performance overhead (maximum of 20 ms). Figure 5.12b illustrates the speedup numbers over one device. The ideal linear speedup shows the ideal scaling speedup with more available devices. As shown, we achieve superlinear speedups. An important parameter in scaling concerns how the *overheads* scale. The superlinear speedup stems from the dramatic reduction of communication overhead as parallelism increases. In traditional data and model parallelism, such overhead increases, which causes sublinear speedup. Figure 5.13 compares latency per image for LCP and model parallelism. On average, LCP models are 3.76x, 8.89x, and 7.17x faster than their model-parallelism counterparts for AlexNet, VGG16, and ResNet-50 (4 and 8 devices), respectively. LCP achieves a maximum and average speedups of 56x and 7x, compared to the originals (Figure 5.28, base bars).

Quantization & Pruning: Techniques that reduce the footprint of DNNs can be applied to each individual LCP branch. Basically, the target output for each LCP branch is now its pre-final activations during optimizations. We study the benefits of lossless quantization and structured pruning on top of our LCP models. Based on our experiment, with 3.13 (<integer.fraction>) quantization, our models do not lose accuracy. Similarly, applying structured pruning [70], for which systolic arrays gain benefits, reduces the size of parameters between 40%–50% per convolution layer without an accuracy drop. Other pruning algorithms increase the sparsity of the data, which is not necessarily beneficial for systolic arrays. Figure 5.28 presents the speedup gained from these techniques normalized

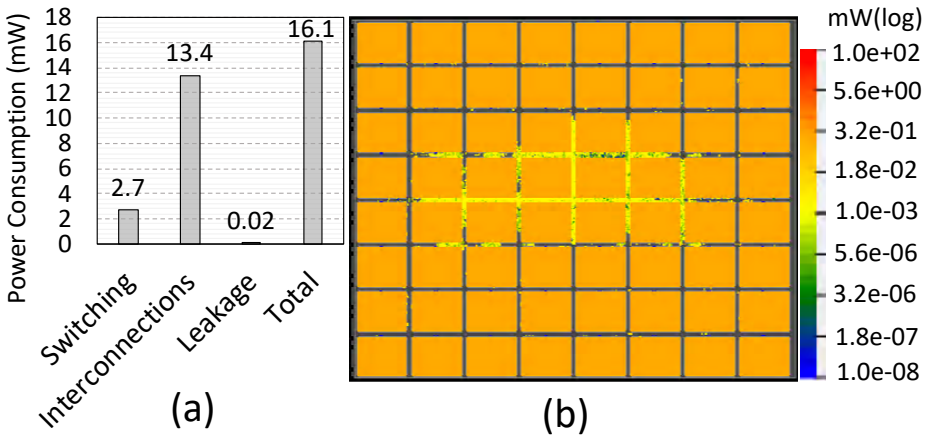


Figure 5.14: **ASIC Power Consumption:** Power consumption for 7-nm ASIC Design @800MHz: (a) breakdown (b) distribution.

to the baseline implementation for each model, the execution performance of which shown in Figure 5.12a. Quantization and pruning themselves, improve the performance of the original models by 1.96x and 2.2x, respectively, and 4.31x when applied together. When quantization and pruning are combined with LCP, the overall performance speedup becomes 14.41x and 16.31x, respectively. Compared to the original models, LCP + quantization and pruning achieves up to 244x speedup (VGG16-split8), and an average of 33x (across all models and variants).

ASIC Implementation

We implement the ASIC design of LCP using an Arizona State Predictive PDK (ASAP) 7nm technology node [137]. Our tool chain includes the Synopsys design compiler (DC) for synthesis, Cadence Innovus for place and route, and Cadence Tempus for timing and power analysis. As an input to our ASIC design, we use our same Verilog code generated by Vivado HLS. Figure 5.5b show the layout of our chip of size 0.107 mm^2 (i.e., 295μm× 365μm). The memory cells shown in the figure represent the FIFO buffers, used for pipelining. Figure 5.14 shows the power consumption of our ASIC design. The breakdown of power consummation leading to a total 16.1 mW is listed in Figure 5.14a. As a comparison point, Eyeriss [87] and EIE [95] consume ≈250 mW and ≈590 mW, respectively. Besides, as Figure 5.14b shows,

power distributes uniformly, which prevents hot spot creation.

5.1.5 Summary

We proposed low-communication parallelization (LCP) models, designed for efficient in-the-edge distribution. LCP models optimize communication while reducing memory and computation by utilizing several narrow independent branches. We presented our results on the accuracy of LCP models. We build a systolic architecture for edge computing both on FPGA and ASIC. Finally, our results on RPis, edge-based FPGAs, AWS instances confirms the benefits.

5.2 Reducing Inference Latency with Concurrent Architectures

In the last section, LCP models illustrate the advantages of models designed with the characteristics of their underlying computational domain in mind. LCP models provided low communication overhead and higher parallelizability. This is because LCP model broke single-chain dependency pattern [1, 37, 42], captures in almost all modern DNNs. Although current platforms exploit parallelism, we discover that, since most architectures capture a *single-chain dependency pattern* [1, 37, 42], shown in Figure 5.15a and Figure 5.15b, we cannot efficiently extend concurrency and distribution beyond current explicit parallelism exposed within intra-layer computations (*i.e.*, matrix-matrix multiplications) to reduce the latency of an inference. In other words, distribution and concurrency, if any, is implemented at data level [148], which only increases the throughput.

The status quo approaches in reducing the inference latency are always applied *after* an architecture is defined (*e.g.*, reducing parameters with weight pruning [15] or reducing computation with quantization [63]). Additionally, for extremely large architectures, limited model parallelism is applied on final layers (*i.e.*, large fully-connected layers that do not fit in the memory [149, 9, 116]). However, since model-parallelism methods do not change the architecture, distributing all layers with such methods adds several synchronization/merging

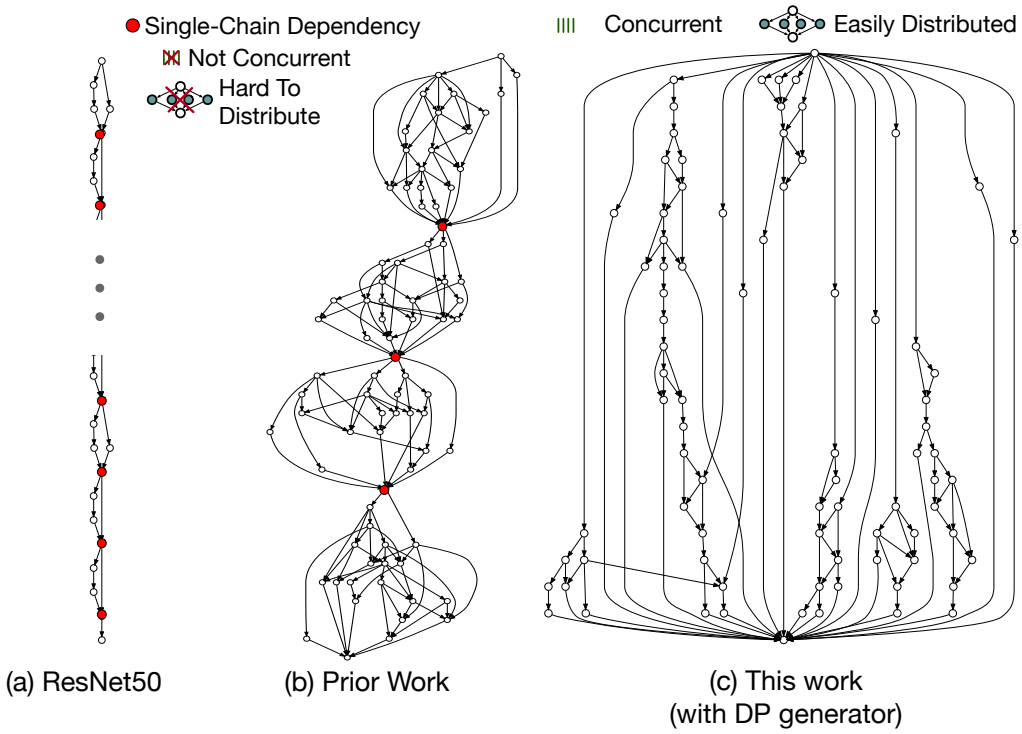


Figure 5.15: **Sampled Architectures Overview** – (a) & (b) Limited concurrency and distribution due to single-chain dependency. (c) Improved concurrent architecture.

points, incurring high communication overheads (Figure 5.15). We discover that the single-chain inter-layer dependency pattern, common in all the well-known architectures and even in state-of-the-art neural architecture search (NAS) studies [105], prevents the efficient model distribution for reducing inference latency.

This last section addresses the single-chain data dependency in current architecture designs and endeavour to inspire discussion for new concurrent architectures. To do so, first, we analyze architectures generated by recent unbiased NAS studies [105] and discover that *scaling/staging* blocks implicitly enforce dependencies. Then, we generate new architectures with prior and our new distance-based network generators using our new probabilistic scaling block. Then, for quantitatively comparing generated architectures, we propose a *concurrency score* that encapsulates important metrics such as communication, load balancing, and overlapped computations, by reformulating the problem as a hypergraph partitioning problem [150, 151]. Based on the scores and experiments, our generated

architectures have higher concurrency and are more efficient for distribution than current architectures, an example of which is shown in Figure 5.15c. Additionally, as shown in Figure 5.16, they provide competitive accuracy while delivering high concurrency, directly proportional to inference latency (Figure 5.22). Our experiment results (on over 1000 samples) show that our architectures achieve 6–7x faster inference time. As an added benefit, the current methods in reducing the inference latency can be applied on top of our generated architectures. The following is our contribution in this section:

- Addressing Single-Chain Data Dependencies: Our concurrent architectures created by network generators (specially the new distance-based generator) break current biased designs by delivering high concurrency.
- Proposing Representative Concurrency Score: Our problem formulation based on hypergraph theory encapsulates critical metrics to quantitatively compare all architectures for efficient distribution and concurrency.
- Conducting Comprehensive Experiments: Our results show that our new models achieves 6–7x in execution performance compared to previous work.

5.2.1 Concurrent Architectures Design

Here, we propose concurrent architectures that break the single-chain dependency pattern for enabling concurrent execution of an inference. To improve distribution and concurrency, we aim to search for an architecture that has minimal communication overhead and is load balanced when it is distributed. To do so, the following provides the general problem formulation. We also extend the representation to quantitatively study distribution and concurrency opportunities, derived by reformulating the problem as a hypergraph partitioning problem.
Overview: The current design of neural architectures is optimized for prediction accuracy and has an implicit bias towards the single-chain approach [105, 100]. This bias limits concurrency and distribution for reducing inference latency. In other words, only the

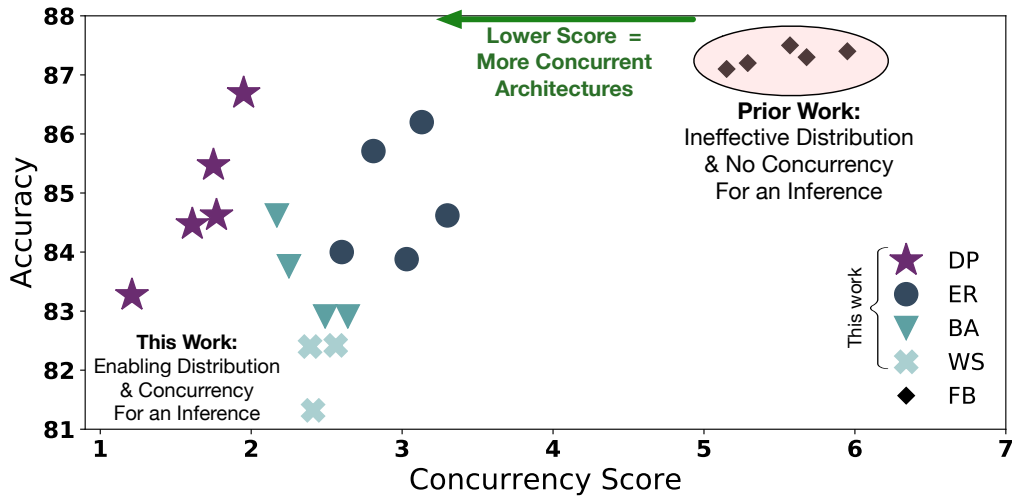


Figure 5.16: **Accuracy vs. Concurrency Score** – Randomly sampled concurrent architectures generated with our NAS consistently achieve competitive accuracies with a higher concurrency and distribution opportunities during an inference (Flower-102).

computation within a layer is performed in parallel and not the computation within a model. To tackle this challenge, we aim to consider concurrency and distribution during the design stage and test if such architectures provide higher concurrency with good accuracy. To do so, first, we use network generators to create a random graph structure, which represents a potential architecture. Among all generated architectures, we sample (without any optimized search) and evaluate generated architectures with our proposed concurrency score. Then, we transform the graph to a DNN and perform experiments. Our final results show a promising direction worth exploring.

DAG Representation: A neural architecture, $\mathcal{N}$, can be represented as a directed acyclic graph (DAG) because the computation flow always goes in one direction without looping. We define a DAG as $\mathcal{G} = (V, E)$ where V and E are sets of vertices and edges, respectively. We define a network generator, f, as a function that constructs random DAG. f creates the edge set, E, and defines the source and sink vertices for each edge, regardless of the type of the vertices. Although network generators could be deterministic (*e.g.*, a generator implemented with NAS approach), we are interested in stochastic network generators. The reasons are two-fold. First, the stochastic generator provides a larger search space than

the deterministic generator, so it is more likely to remove any bias. Second, since, unlike prior work, we don not use scaling/staging to glue different parts of our NAS generated network [105] (shown in Figure 5.15b), stochastic generators provide more options for a potential solution. Note that the generated DAG only represents the dataflow and does not include the weights, which are learned in subsequent steps.

DAG to DNN: Once we have found a promising DAG representation after the concurrency score study, we transform the DAG into an actual DNN. Vertices in DAG are components (*e.g.*, layers or sub-networks) and edges are connections. Within the process of transformation, we convert the nodes in DAG to a block of layers and connect blocks with its corresponding edge in DAG. Each vertex, V_i, has several properties such as type of the layer and its properties (*e.g.*, depth, width, activation size, *etc.*). In this paper, we use a uniform computation in vertices: ReLU, 3x3 separable convolution [39], and batch normalization [142].

5.2.2 Network Generators

We use three classical random graph generators as baselines. Additionally, after discovering that state-of-the-art generators do not generate a concurrent architecture, we propose a new graph generator with distance-based heuristics. Below, we describe the generators identified by how their stochastic nature influences the graph. Note that although the first three generators are based on [105], to generate concurrent architectures, we have removed the introduced staging blocks, which enforces the single-chain dependency in prior work. Thus, all the studied architectures in this work are novel and have never been studied before.

Once we obtain an undirected random graph from the generator, we convert the undirected graph to DAG by using the depth-first search algorithm. The vertices with smaller vertex ID is traversed earlier than vertices with larger ID. As the final step, we add an input vertex to all vertices without predecessors and an output vertex to all vertices without successors. This ensures that we obtain a DAG with a single source and sink.

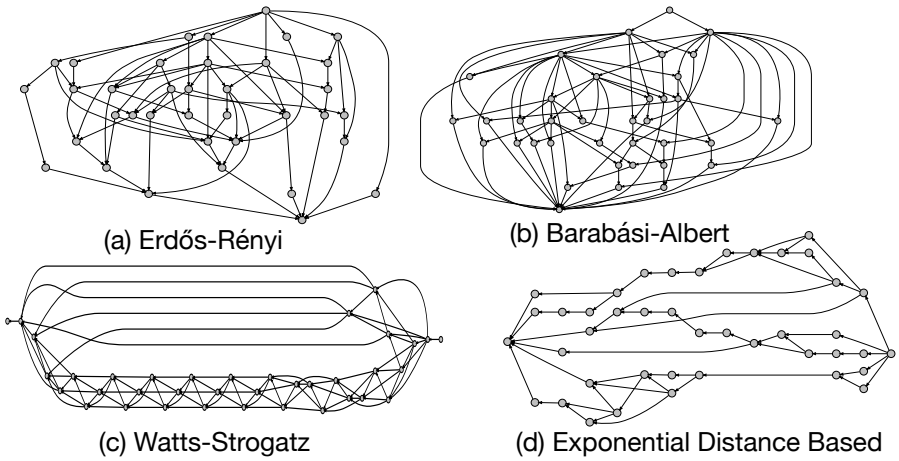


Figure 5.17: **Network Generators** – Four examples of different random graph generators. Note that only (d) produces a good concurrent balanced graph.

(1) Independent Probability: In this group, the probability of adding an edge is independent of other properties. Similar to the Erdős and Rényi model (ER) [152], in which an edge exists with a probability of P. Generators with independent probability completely ignore the graph structure and create a connected graph (Figure 5.17a) that is hard to efficiently distribute.

(2) Degree Probability: In this group, the probability of adding an edge is defined by the degree of one of its connected vertices. A vertex with a higher degree has more probability of accepting a new edge. Figure 5.17b shows an example of such a generator. Barabási-Albert model (BA) [153], first adds M disconnected vertices, then for the total number of vertices until N, it adds a total of M edges with a linear probability proportional to the degree of each vertex (*i.e.*, a total of $M(N - M)$ edges). Generators with degree probability create a tree-structured graph, in which at least one vertex is strongly connected to other vertices. Such a graph structure is hard to distribute since all the vertices are dependent on at least one vertex, if not more.

(3) Enforced Grouping: In this group, initially, a pre-defined grouping is performed on disconnected vertices and then edges are added based on the groups. Small world graphs [154, 155, 156] are good examples. In one approach (WS) [155], vertices are placed

in a ring and each one is connected to $K/2$ neighbors on both sides. Then, in a clockwise loop on vertices, an existing edge between its i_{th} neighbor is rewired with a uniform probability of P for $K/2$ times. As shown in Figure 5.17c, a graph with WS algorithm tends to form a single-chain structure if P is small. With a larger P, the structure becomes similar to ER.

(4) Distance Probability: In distance probability (DP), initially, a pre-defined grouping is performed on disconnected vertices, then a distance probability function defines the existence of an edge. We first arrange the vertices in a ring. Then, the probability of adding an edge between two vertices is dependent on their distance. In other words, closer vertices have a higher probability of getting edges.

– *Distance Metrics:* We define distance d as the smallest number of nodes plus one between two nodes in a ring. The maximum distance can be half of the total number of nodes, which is $N/2$. We use the distance to re-scale the passed in probability P presented in WS. We use exponential re-scaling function:

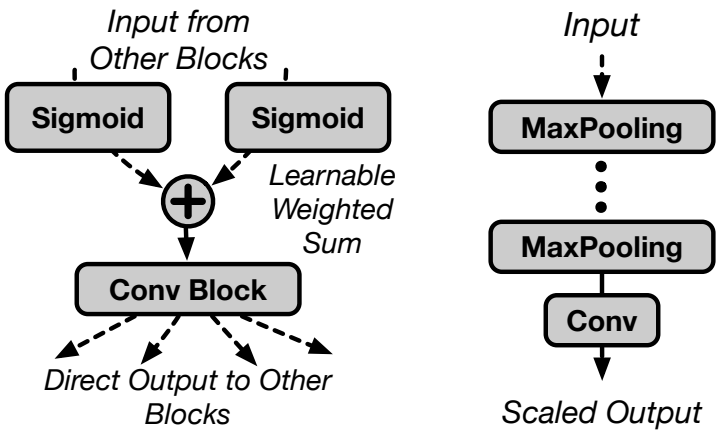
$$P_{\text{new}} = \alpha P^{\beta d}, \tag{5.1}$$

in which α and β are constants. The probability quickly decreases as the distance increases. This mechanism naturally creates multiple locally strongly connected graphs, Figure 5.17d, which can be distributed.

5.2.3 Transformations

Transformations are operations, the main objective of which is to create a reasonable architecture, that happens after the construction of the DAG. We first introduce the building blocks, which include a scaling building block that, contrary to previous work, does not enforce a single-chain dependency.

Building Block: During the process of transforming a DAG to DNN, vertices are interpreted as basic building blocks, as shown in Figure 5.18. Inside a basic building block, Sigmoid activations are applied on inputs, then, the activations are summed with a learnable weighted sum. The Sigmoid function is used to avoid weighted sum overflow. As described before,



(a) Basic Building Block. (b) Scaling Building Block.

Figure 5.18: **Building Blocks** – Building blocks used for conversion from DAG to DNN.

the conv block consists of a ReLU, 3×3 separable convolution, and batch normalization.

Redefining Staging: Staging is deemed to be necessary for all NAS generated architectures to reduce the computation and facilitate learning. For staging, after a few layers, usually, the common method is to gather and merge outputs from all transformation vertices, conduct downsampling, and channel upsampling. However, such staging points create a rigid architecture with single-chain dependencies that are hard to distribute and execute concurrently (*e.g.*, [105]). To address the single-chain bottleneck problem caused by staging, the first solution is implementing a uniform channel size for the entire architecture. In other words, all conv blocks share the same filter size. Thus, there would be no need to merge and synchronize at a point during an inference. However, as shown in Table 5.5, the uniform channel size approach works well on a small image dataset (*e.g.*, Cifar-10), but it fails to achieve good accuracy on a dataset with larger image dimension (*e.g.*, Flower-102).

In this paper, we propose individual staging after any conv block. Because of that, inputs to a conv block could have different dimensions. To tackle this problem, we dynamically add a new scaling block in the process of construction. The scaling block consists of a number of maxpooling layers. Maxpooling layers downsamples the dimensions to match with the smallest dimension in the input. We also use 1×1 convolution layers to upsample

the channel size to match the highest channel size in the inputs in these scaling blocks. Therefore, we avoid bottlenecks in generated architecture.

We adopted two design choices for the staging mechanism. In the first design, greedy-based staging, we start with greedy-based staging. Within the construction process, we set an upper limit for channel size. As long as channel sizes have not reached the upper bound, we conduct staging (*i.e.*, downsample the input & upsample the channel). However, this design raises an issue that intermediate outputs are quickly squeezed through the maxpooling layer, which discards important features. This approach hurts the accuracy to some extent. s In the second design, probabilistic-based staging, we use a probabilistic method in staging. In this design, although the channel size may have not reached the limit, staging is done with a fixed probability of 0.5 to avoid discarding features too quickly. As shown in Table 5.6 and Table 5.7, the probabilistic approach achieves better accuracy rate than the greedy-based approach. In addition, Table 5.7 shows that probabilistic staging supports higher accuracy with less parameter size because (i) probabilistic staging gracefully discards features, so the architecture learns better; and (ii) the aggressive greedy-based staging creates more size mismatch, so it requires more scaling blocks.

5.2.4 Concurrency & Distribution

Our goal in this paper is to inspire concurrent architecture designs to improve inference latency performance. As a result, besides common accuracy consideration, we need to study concurrency and distribution opportunities of a candidate architecture. To help the

Dataset	**Baseline**	**DNNs with Uniform Channels**
Cifar-10 32×32	80.70	81.13
Flower-102 224×224	87.80	74.73 **(Fails to Scale!)**

Table 5.5: **Accuracy of Uniform Channels** – The mean accuracy comparison between sampled group architectures with uniform channel *vs*. handcrafted without any advanced optimizations. (baselines Cifar-10 and Flower-102 are vanilla CifarNet and ResNet-50, respectively).

Staging/Samples	A	B	C	Overall Mean
Greedy	82.30	81.32	82.42	82.01
Probabilistic	82.42	86.69	84.62	**84.58**

Table 5.6: **Average Accuracy** – Comparison of randomly sampled group of generated architectures with different staging choices (trained on Flower-102).

Staging/Samples	A	B	C	Overall Mean
Greedy	2.31	2.27	2.63	2.40
Probabilistic	3.00	3.28	3.58	**3.29**

Table 5.7: **Average Accuracy/Parameters Ratio** – Comparison of randomly sampled generated architectures with different staging choices (trained Flower-102).

community to extend our study, instead of focusing and showcasing on a single architecture, we are interested in finding a customized *concurrency score* (CS) for a given architecture, $\mathcal{N}$, that is easily calculated. In this way, we can study various architectures and future works that can further improve this work. CS shows how optimal the concurrent and distributed task assignment for an architecture is. Lower PS score represents fewer communications, better load-balanced tasks, and more distribution opportunities with more overlapped computation, so the architecture is more efficient for concurrency.

Metrics in The Score: We can formulate our problem of allocating tasks on n units as a multi-constraint problem. The first constraint is that all units should perform the same amount of work, or be load balanced. Second, the communication amount, the main bottleneck in distribution, should be at a minimum. And third, we want to minimize runtime by increasing overlapped computations among the units. The first two constraints are addressable by finding a set of hypergraph partitions, in which we divide the vertices into equally weighted sets so that few hyper-edges cross between partitions. The derivable metric is the amount of variability in loads (δ_W) and a total of communication (Λ). The third constraint is measurable by finding the longest path between the input and output vertices on the DAG and quantify concurrency (η). For instance in pipeline parallelism, the longest path is the entire architecture, as a result the latency is never reduced (and throughput is increased). Now, we provide the formal definition of these solutions by first studying the

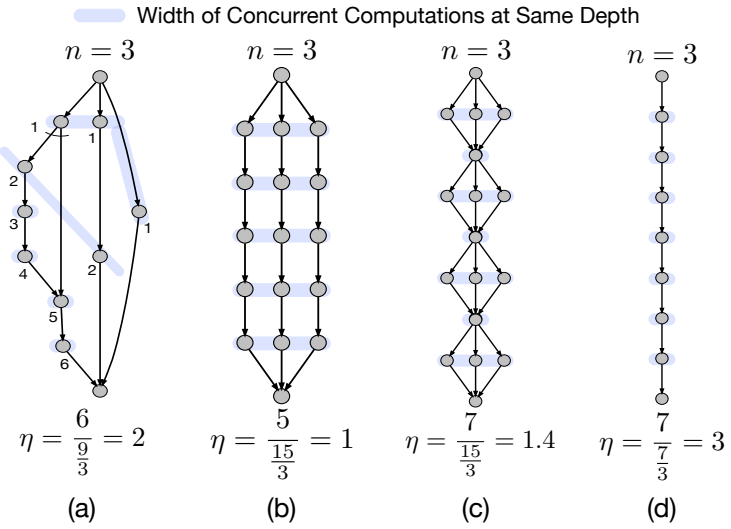


Figure 5.19: **Overlapped of Computation Metric** – Illustration of η.

DAG.

Maximizing Overlapped Computations: We measure how overlapped is the inter-layer computations of an architecture from its DAG, or η, as a ratio. We measure this by observing the longest path in the distinct paths between input and output vertices in the DAG, $\mathcal{G}$, relative to the number of the computation cores, n. Assume $\{d_i\}$ is the set of distinct longest paths in $\mathcal{G}$. We define η as

$$\eta = \frac{\max\{d_i\}}{|\mathcal{V}|/n}, \tag{5.2}$$

in which $|\mathcal{V}|$ is the total number of vertices. Figure 5.19 depicts an examples of η. A higher η value shows a more limited opportunity to overlap the computation. Figure 5.19 also shows the width of the overlapped computation at the same depth (*i.e.*, DFS depth with the source of input), which is a good representation of why some architectures are more efficient for concurrency.

Hypergraph Representation: Using graph representations in task assignment for distributed computing is a well-known problem [157]. Basically, in the generated DAG, vertices of the graph represent the units of computations, and edges encode data dependencies. We can indicate the amount of work and/or data, by associating weights (w) and costs (λ) to vertices and edges, respectively. However, a DAG representation does not sufficiently

capture the communication overhead, load balancing factor, and the fact that some edges are basically sending the same data/features. Therefore, for task assignment, we use an alternative graph representation, derivable from the DAG, hypergraph. A hypergraph [150] is a generalization of a graph, in which an edge can join any number of vertices [158]. The hypergraph representation, common in optimization for integrated circuits [151], enables us to consider the mentioned factors.

Formal Definition of Hypergraph: A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is defined as a set of vertices $\mathcal{V}$ and a set of hyper-edges $\mathcal{E}$ selected among those vertices. Every hyper-edge $e_j \in \mathcal{E}$ is a subset of vertices, or $e_j \subseteq \mathcal{V}$. The size of a hyper-edge is equal to the number of vertices.

Hypergraph Partitioning: We assign weights (w_i) and costs (λ_j) to the vertices ($v_i \in \mathcal{V}$) and edges ($e_j \in \mathcal{E}$) of the hypergraph, respectively. $\mathcal{P} = \{V_1, V_2, V_3, ..., V_P\}$ is a P-way partition of $\mathcal{H}$ if (i) $\forall V_i, \emptyset \neq V_i \subset \mathcal{V}$, (ii) parts are pairwise disjoint, and (iii) $\bigcup \mathcal{P} = \mathcal{V}$. A partition is balanced if $W_p \leq \varepsilon W_{\text{avg}}$ for $1 \leq p \leq P$, where $W_{\text{avg}} = \sum_{v_i \in \mathcal{V}} w_{v_i}/P$ denotes the weight of each part, and ε represents the imbalance ratio, or δ_W.

In a partition $\mathcal{P}$ of $\mathcal{H}$, a hyper-edge that has at least one vertex in a part is said to connect that part. The number of connections γ_j of a hyper-edge e_j denotes the number of parts connected by e_j. A hyper-edge is a cut if $\gamma_j > 1$. We define such hyper-edges as an external hyper-edges $\mathcal{E}_E$. The total communication for $\mathcal{P}$ is

$$\Lambda = \sum_{e_j \in \mathcal{E}_E} \lambda_j (\gamma_j - 1). \tag{5.3}$$

Therefore, our two constraints can be defined as a hypergraph partitioning problem, in which we divide a hypergrpah into two or more parts such that the total communication is minimized, while a given balance criterion among the part weights is maintained. We can solve this NP-hard [151] problem with multi-paradigm algorithms, such as hMETIS [159] relatively fast. Note that solving this problem is a pre-processing step, which does not affect runtime.

Concurrency Score: Now, we have the tools to calculate the concurrency score, CS. Figure 5.20 summarizes all the steps to derive our metrics: Load variability, δ_w; total amount

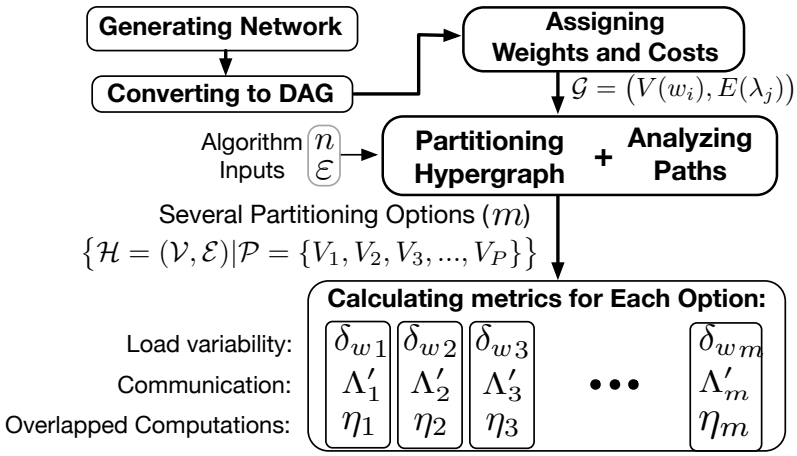


Figure 5.20: **Calculating Concurrency Score** – Summarizing steps for deriving the score.

of communication, Λ; and overlapped computations, η. Hypergraph algorithm accepts the number of units and a higher bound of ε. By changing the ε, we create a set of partitioning options, for each of which we compute all the metrics. Note that the DAG input requires a weight and cost value for every vertex and edge, respectively. Both of these values are easily derivable. The weight of a vertex is directly proportional to its floating operations (FLOPs), reported by most frameworks. The cost of an edge is directly proportional to the transferred data size. To get CS, first, we need to normalize the communication metric. We write Λ as $\Lambda' = {}^{\Lambda}/_{(U_c \times n)}$, in which U_c is a unit of data and n is the number of units. We define

$$\text{CS} = \sqrt[1/3]{\delta_w^a \Lambda'^b \eta^c}, \tag{5.4}$$

as a custom concurrency score, in which a, b and c are constant that show the relative importance of each metric for a user. In this paper, we assume $a = c = 1$ and $b = 1.5$, for a higher priority for communication. We chose U_c as the smallest amount of communication for an edge in a generator. Hence, a higher CS value shows poor distribution and concurrency opportunities.

5.2.5 Concurrent Architectures Experimental Studies

In this section, we evaluate our generated architectures by comparing our customized generator and transformation process with prior work. The results demonstrate that our

generated architectures preserves accuracy while achieving better concurrency scores by removing the implicit bias of single-chain dependency. Besides, by running the final architecture on actual devices, we show that the concurrency score provides reasonable heuristic about the real performance.

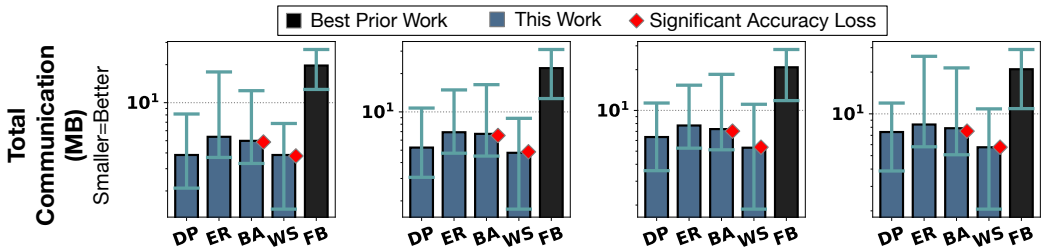
Experimental Setup

Generators: All generators use probabilistic scaling blocks. FB represents prior work in unbiased NAS with staging blocks [105]. As mentioned in subsection 5.2.2, although ER, BA, and WS generators are based on [105], we remove the staging block that causes the limited concurrency. As a result, all the studied network generators and resulted architectures are novel and have never been studied before.

Randomization: To evaluate the accuracy of randomly generated architecture, we collect representative samples with *no optimized search*. we followed the same training procedure for architectures and reported the average accuracy. For CS, total communication, and computation time evaluations, we collect 1,000 samples with no optimized search and compare across different generators.

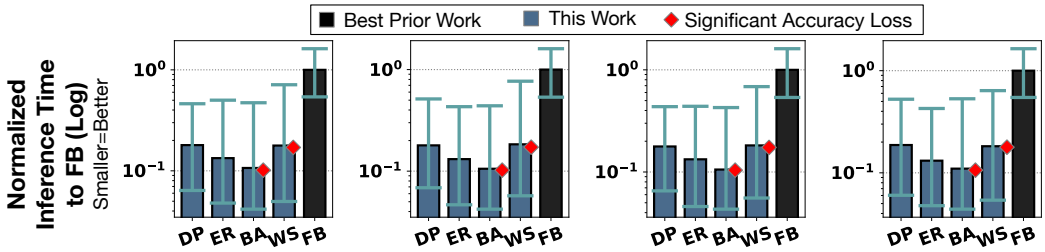
Datasets: We conducted experiments on multiple datasets to ensure the extensibility of concurrent architectures. We use two image classification datasets; (i) Cifar-10 [160], which contains 60K 32×32 images in 10 classes; and (ii) Flower-102 [161], which contains 16K 224×224 images in 102 classes. We strongly encourage future extensive studies on larger datasets, but given the heavy-compute bound of NAS-based experiments, we chose to use representative datasets studied in most of the prior works [162].

Training Procedure: We use a uniform training pipeline with a stochastic gradient descent optimizer for all architectures. We train on Cifar-10 with 100 epochs and on Flower-102 with 300 epochs. We report the top-1 classification accuracy on the test sets. For the first 100 epochs, we set the learning rate to be 1e-3 and momentum to be 0.9. We changed the learning rate to 5e-4 and momentum to 0.95 for the remaining 200 epochs on Flower-102.



(a) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 4$ (b) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 6$ (c) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 8$ (d) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 10$

Figure 5.21: **Total Communication with Distribution** – Measured communication in MB for 1000 sampled architectures in each category for 40 vertices on {4,6,8,10} units.



(a) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 4$ (b) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 6$ (c) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 8$ (d) $|\mathcal{V}| = 40$, $|\mathcal{P}| = 10$

Figure 5.22: **Inference Time** – Normalized inference time normalized to FB (subsubsection 5.2.5) for 1000 sampled architectures in each category for 40 vertices on {4,6,8,10} units.

Implementation: We implemented all graph representations in Python NetworkX [163] library. Then, we convert a graph to a PyTorch [164] compatible model. We constructed a graph-based forwarding path in PyTorch module class to directly reproduce the graph structure.

Experiments

We analyze the results from three perspectives, communication, latency, and concurrency score. Because we are interested in finding a general solution, we start with the architecture stability evaluation that particularly focuses on the architecture parameter size. Then, we show the generated architectures achieve competitive accuracies, while, in the last part, we illustrate the high concurrency and distribution opportunities of these architectures.

Architecture Stability: For the architecture stability experiment, we used a fixed number

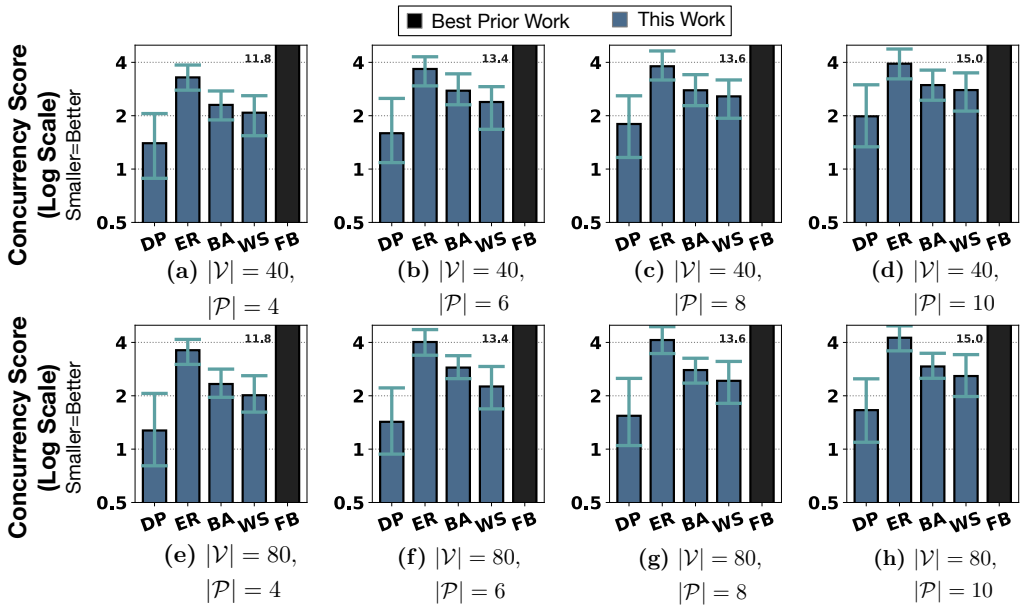


Figure 5.23: **Concurrency Scores** – Measured CS for 1000 sampled architectures in each category with {40,80} vertices on {4,6,8,10} units (subsubsection 5.2.5).

of 40 building blocks. We created 1,000 samples from each network generator. We recorded mean and standard deviation regarding the parameter sizes. We also evaluate the architecture stability under different staging design choices (greedy vs probabilistic). From Table Table 5.8, we see that proposed generators with greedy scaling blocks creates larger but more stable architectures than with probabilistic scaling blocks. Additionally, we see that our proposed DP generator creates the most efficient architecture. We will see that architectures that use DP generators are generally the most optimized.

Accuracy Study: Here, we demonstrate that the concurrent architectures achieve competitive accuracy on both Cifar-10 and Flower-102 datasets. Given the heavy-compute

		ER	AB	WS	DP
Greedy Staging	**Mean**	48.63	48.33	42.03	35.03
	Std	1.11	0.91	1.28	2.25
Probabilistic Staging	**Mean**	46.03	45.63	36.44	26.69
	Std	2.70	4.41	3.52	3.05

Table 5.8: **Parameter Size Stability** – The mean and standard deviation of parameter size in sampled generated architectures with different staging.

	Mean Acc.	Best Acc.	Mean Acc./Param.	Best Acc./Param.
CifarNet	80.70	80.70	5.38	5.38
ER	81.33	81.81	4.94	5.03
BA	80.29	81.66	4.81	4.92
WS	79.89	81.45	4.75	4.84
DP	80.87	82.47	4.81	4.90

Table 5.9: **Concurrent Architectures on Cifar-10** – Overall sampled metrics.

	Mean Acc.	Best Acc.	Mean Acc./Param.	Best Acc./Param.
ResNet-50	87.80	87.80	3.43	3.43
ER	84.88	86.20	2.11	2.43
BA	82.91	84.62	2.41	2.91
WS	81.46	86.57	3.17	3.10
DP	84.66	86.69	3.19	3.28

Table 5.10: **Concurrent Architects on Flower-102** – Overall sampled metrics.

bound of NAS-based experiments, we encourage further studies on larger datasets. We used the same architecture samples as before without any optimized search and reported both mean and best results. As shown in Table 5.9 and Table 5.10, our concurrent architectures achieve comparable accuracy on both datasets. Generated DNNs achieve better or similar accuracy on Cifar-10. For Flower-102, because both network generation and transformation processes have more randomness, the mean accuracy has a small gap compared to the baseline. However, the best accuracy is close to the baseline, so we believe the accuracy gap can be leveraged by conducting an optimized search in terms of accuracy.

Concurrency Study: Finally, to show improved distribution and concurrency opportunities, we examined the concurrency score of our architectures to ResNet-50 and FB (subsubsection 5.2.5) by sketching width/depth histograms in Figure 5.24. As shown, we achieve higher width/depth, which enables more concurrency, while provides lower maximum depth, which enables shorter execution time. To quantitatively compare the generators and FB, Figure 5.23 depicts concurrency scores, summarized on over 1000 architectures in each category per set. As seen, our generators (and specifically DP) consistently gain the best score. Moreover, to gain more insights, Figure 5.21 and Figure 5.22 illustrate total communication

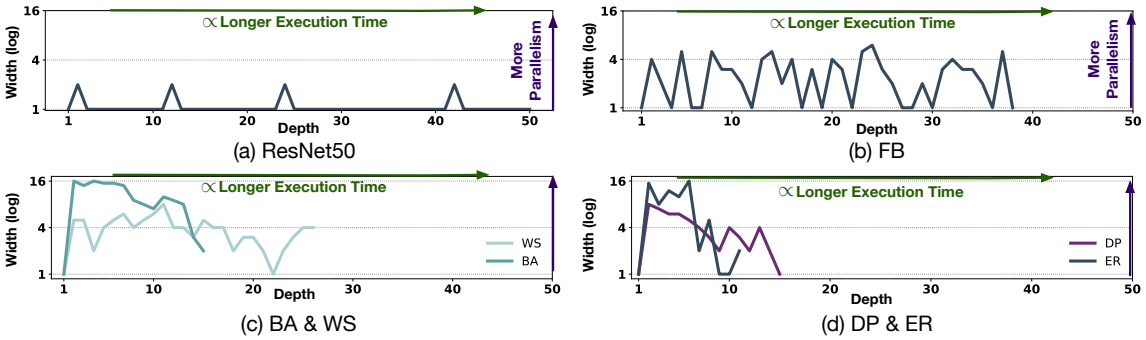


Figure 5.24: **Width/Depth Histograms** – Illustration of ResNet50, FB, and concurrent architectures, which enable more concurrency and shorter inference latency.

with distribution and inference (*i.e.* computation) time, when each architecture is deployed on $|\mathcal{P}|$ units. We see that though ER and BA methods deliver better computation speedup, they suffer performance slow down more from data communication. For our new generator, DP, we see an 6–7x speedup in inference time. We observe a close relationship between the reported score and actual latency and communication. In fact, latency and communication measure performance in an orthogonal way, but CS score captures the overall efficiency of the generated architecture pretty well and could be used in future studies.

Distribution

To distribute the generated networks according to the number of units, we first group node in the same sequential path together to minimize the communication overhead. After the nodes in the graph are grouped together, we use a heuristic-based greedy algorithm to distribute all nodes to units. The objective of the algorithm is to balance the workload. To make the load balancing simple, we assume the final goal is that each unit performs a similar amount of computations. Ultimately, this process can be improved using various other techniques that currently is out of the scope of this paper. Here, we provide an example of our process, which starts from network generation to workload distribution.

Network Generation: Figure 5.29 demonstrates a example of raw random neural network generated. This network is later fed into a grouping and distribution algorithm to decide

which unit runs which nodes.

Distribution to 2,4 and 8 Units: Figure 5.29 shows network distribution on 2,4 and 8 units. The coloring marks the node is distributed on which unit. Because all units need to run the computations of the first node, we leave it as a common node (this could be just a scatter operation). In addition, for the last node, an extra unit is needed to merge all results together, so we mark that unit as black (this could be just a gather operation).

Load Balancing: From the graphs, we observe that the current grouping and distribution algorithm does well load balancing under the scenario with a small number of units. The quality of load balancing affects the final inference latency, because the final results may slow down due to a bottleneck node, which happens when unbalanced loads exist. We conduct a load balance quality study as well as shown in Figure 5.25. We use normalized Shannon entropy value to indicate the load balancing quality (the higher the number represents the load is more balanced, and 1 means the load is perfectly balanced across distribution units). In the Figure 5.25, we showcase the median, 25% – 75% percentile, and 1% – 99% percentile load balancing qualities. We observe that as the number of distribution units increases, the overall load balancing quality downgrades and the variation of quality increases. We aim to develop distribution algorithms with higher quality; however, currently, our aim in this paper is showing that parallel inference computations of a single request is a viable option

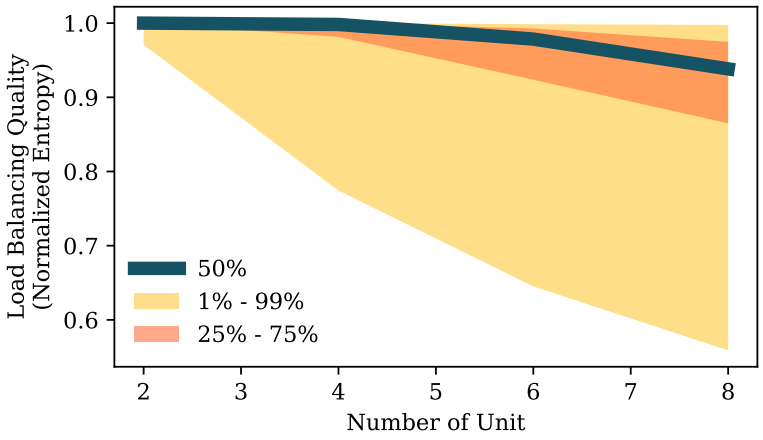


Figure 5.25: **Load Balance Quality** – The load balance quality analysis on two, four, six and eight units compared to the normalized Shannon entropy value.

and should be studied more.

Performance Scaling: As the final step, we also conduct a study on performance scaling. We use a total of 10 AWS t2.micro EC2 instances for performance evaluation. Each instance is equipped with only 1 vCPU and 1 GB memory. The specification are chosen to emulate edge units with limited compute and memory that have a higher computational cost (remember that constants in the Equation Equation 5.4 give higher priority to communication). As shown in Figure 5.26, the inference latency improves when the system has more distribution units. However, The latency stops to decrease as the number of distribution units becomes 8, because the workload is not well balanced on each unit, as shown in our load balancing study. In this example, the bottleneck unit in the system causes longer latency for the entire system.

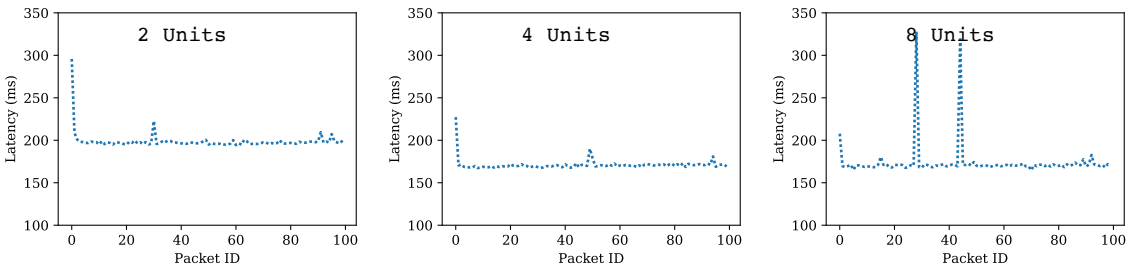


Figure 5.26: **Performance Scaling** – the random neural network latency on two, four, and eight distribution units.

5.2.6 Summary

We proposed concurrent architectures that break the single-chain of dependencies, a common bias in modern architecture designs. We showed that these architectures are concurrent and have more distribution opportunities for reducing the inference time while achieving competitive accuracy. Since we discover that previous NAS studies were implicitly biased in creating a sequential model, we introduced a new generator that naturally creates concurrent architectures. To quantitatively compare concurrent architectures, we proposed the concurrency score that encapsulates critical metrics in distribution.

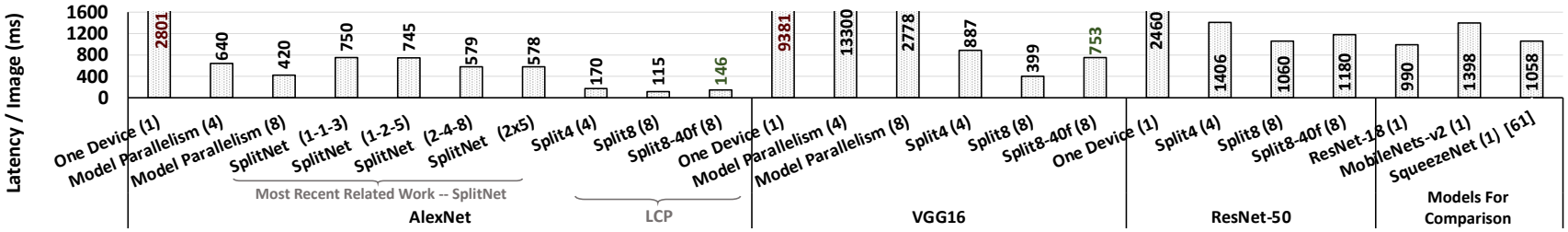


Figure 5.27: **Latency per Image:** Model-parallelism, SplitNet [80], and LCP models on RPi (number in parenthesis is #devices).

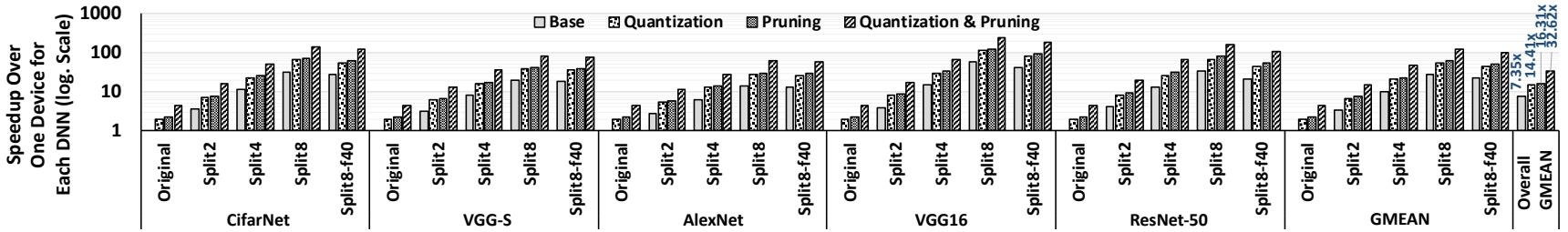


Figure 5.28: **Applying Quantization and Pruning on LCP:** Edge FPGA with tailored hardware speedup with quantization & pruning. Additional speedup is gained by applying lossless ($\leq 0.1\%$) quantization and structured pruning.

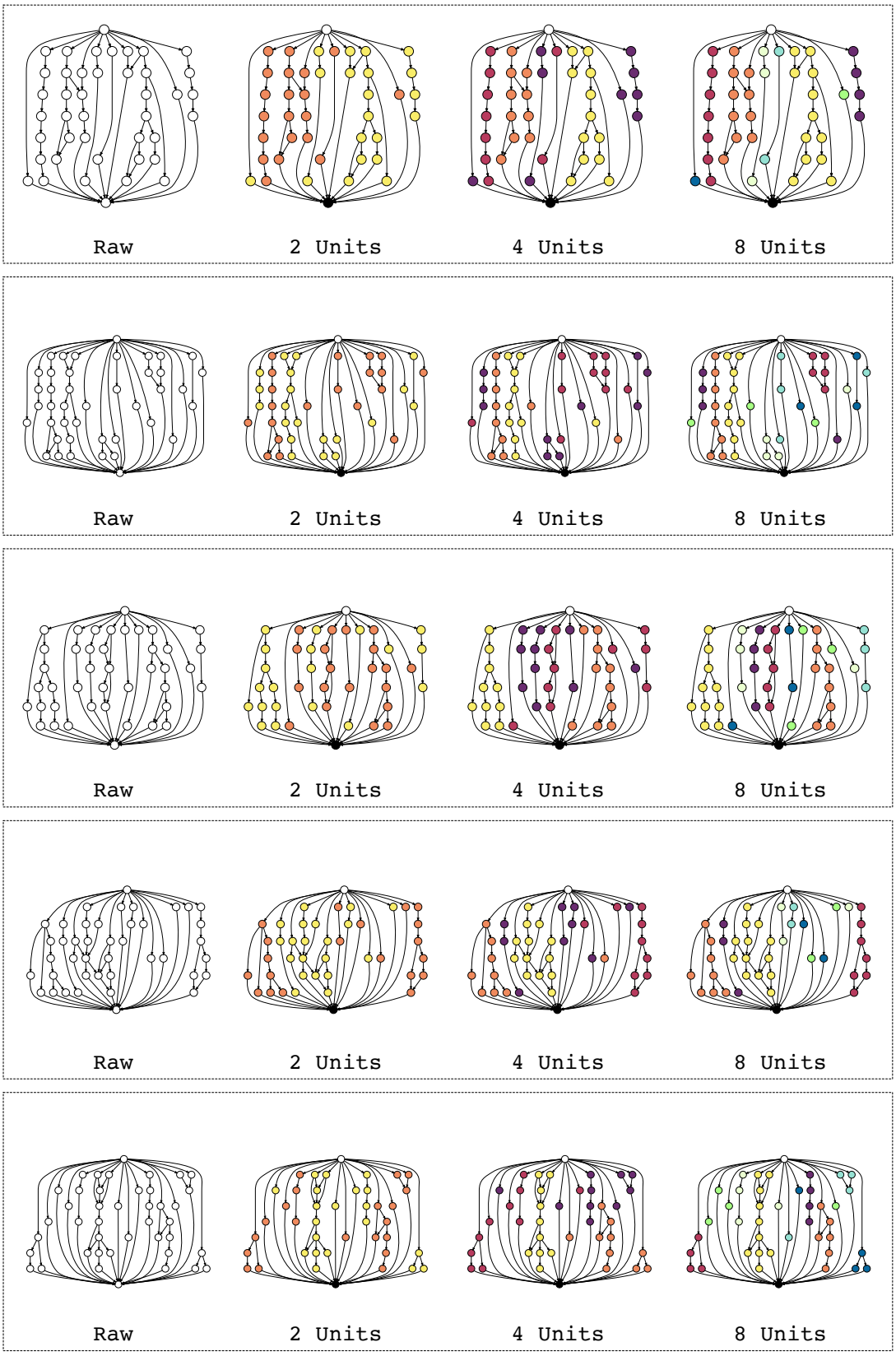


Figure 5.29: **Random Neural Network Distribution** – This gives 5 examples of raw random generated neural networks, their distributions on two, four and eight units.

CHAPTER 6

INCREASING THE RELIABILITY OF DISTRIBUTION

Distribution the computation on edge devices bring another challenge, reliability. This is because the distribution is susceptible to any failures, from short disconnectivity to losing a device. Thus, we may lose valuable time-sensitive and real-time information. With these intermittent or permanent failures, we may lose valuable time-sensitive and real-time information. Thus, the robustness of the entire system becomes a concern, specifically when users may rely on this system for many sensitive and time-critical applications. Moreover, the robustness issue is exacerbated by the local wireless networks of these systems, causing the latency between the devices to be unreliable and unstable. This fact necessitates developing a robust method for tolerating these failures and helping with long latencies.

To do so, first, we analyze general methods of distributing the computations of DNNs (with a focus on convolution neural networks (CNNs)) and how their underlying matrix-matrix computations are affected by distribution. Such a detailed study is necessary to introduce a general seamless method within the underlying library or machine learning framework. Then, we propose a new recovery method based on coded distributed computing (CDC) that enables distributed DNN models on edge and edge devices to tolerate failures and not lose time-sensitive and real-time information. Our method is inspired by CDC applications in big data analytics [106, 165], and speeding up distributed learning using codes [114]. In summary, these works *theoretically* analyze CDC methods that reduce latency by increasing computation (See related work, section 3.6).

To introduce robustness in distributed edge systems, we introduce an extra coded computation per device. The introduced extra computations are derived by thoroughly studying various distribution techniques in their underlying matrix-matrix computations for inference operation in DNNs. These extra computations are similar in nature to those of DNNs,

which ease balancing the work among edge devices and reduce the deployment cost. Such a balanced distribution is essential in attaining the expected performance. Additionally, since our method is implemented in the underlying matrix-matrix computations of DNN layers, it does not require extensive changes to the user's program and is implemented at the library level. Moreover, our method, even at the time of failures, provides close-to-zero recovery time, which is necessary for critical time-sensitive tasks. This is in contrast with approaches that sacrifice latency for robustness to recompute the missing part of the data. Finally, compared with conventional modular redundancy methods, that introduce redundancy in computation by introducing a linear number of additional devices, our method has a constant cost with the increasing number of devices. We demonstrate our method on distributed systems comprising of Raspberry Pis (RPis), which represent the de facto choice for several small and edge use cases.

In summary, this is the first work [129, 166], to the best of our knowledge, that improves robustness in distributed edge systems, with the following contributions:

- We thoroughly analyze how general methods of distributing the computation of DNNs affect the underlying matrix-matrix computations.
- We propose a novel fault recovery method based on CDC that has close-to-zero recovery latency, does not disturb the balanced work assignment in distribution, requires minimal changes to the user's program, and has a constant cost with the increasing number of devices.
- We demonstrate our method on distributed systems of up to 12 Raspberry Pis and report our experimental results.

6.1 Reliability Importance

To understand why reliability is important let us dive deeper in the communication challenge presented in subsection 2.3.4 and illustrate unreliability in the communication latency of

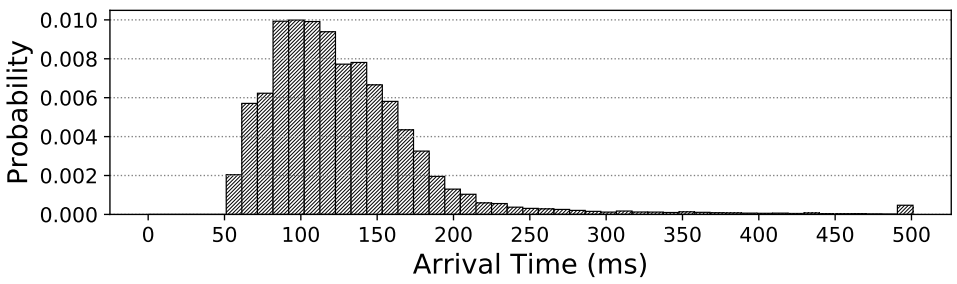
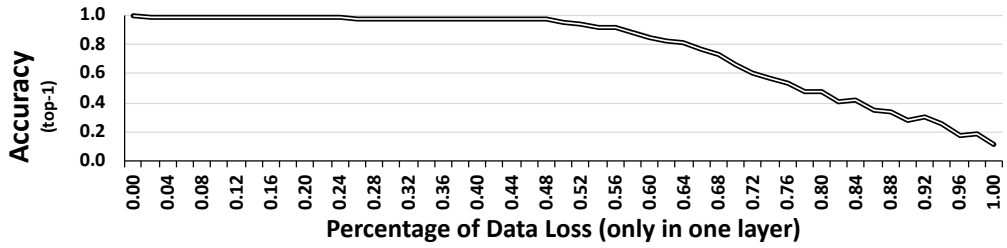


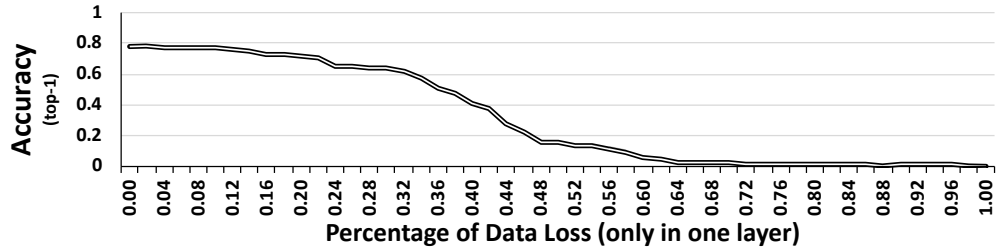
Figure 6.1: **Packet Arrival Time Histogram:** Arrival time histogram of data packets in a WiFi network for a four-device edge system with RPis.

edge systems, Figure 6.1 shows a histogram of the arrival times for data packets in a four-device edge system with four RPis [41]. This system performs the computation for a fully-connected layer of size 2048 in a distributed fashion and waits for the response. The measured time for the computation of a fully-connected layer of size 2048 on a single device is 50 ms. This is why, in Figure 6.1, no packet arrives earlier than 50 ms. As seen, around 34% of the arrival times is within 100 ms, and 42% is within 150 ms. So, even after 2x the computation time, around 34% of the packets have not arrived yet. Such behavior in distributed systems causes *straggler problem*, in which the slowest node in the distributed system defines the total latency. Our method, by introducing robustness in such systems, can additionally alleviate the straggler problem while also guaranteeing close-to-zero recovery latency.

To understand how failures are destructive in DNN applications, we perform another set of experiments, in which some part of data within a layer is lost. We choose two models: LeNet-5 [54] and Inception v3 [167]. LeNet-5 is a simple model that detects handwritten digits from 10 classes and consists of only five layers. On the other hand, Inception v3 is a modern DNN model for image recognition for 1k classes with 159 layers. Figure 6.2 illustrates the accuracy drop in these models when some part of the data in a layer is lost. As seen, for large percentages of data loss (> 70%) per layer that are common in distributed edge systems, the accuracy drop is destructive. Additionally, by comparing Figures Figure 6.2a and b, we see that that the sensitivity to data loss in more generalized models will only



(a) 10-class digit recognition (LeNet-5)



(b) 1000-class image recognition (Inception v3)

Figure 6.2: **Effect of Data Loss on Accuracy:** High percentage data loss, common in distributed edge systems, causes destructive accuracy drops.

become worse. Since the amount of data loss happens in larger granularities, the current robustness methods in DNNs (e.g., bit-level tolerance [168]) are insufficient to recover the loss. In contrast, our proposed robustness method is designed specifically for such a high amount of data loss and can recover from it with close-to-zero latency.

6.2 Robustness with CDC

This section first describes how distribution methods presented in Chapter 4 change the computation of each device from the view of underlying matrix-matrix computation using the background information provided in Chapter 2. Such analysis helps us to easily generalize our method and apply it at the library level. Next, we provide a simple example of our robustness method that handles only one output per device. Then, we generalize our method to multiple outputs per device. Finally, we study the suitability of the distribution methods.

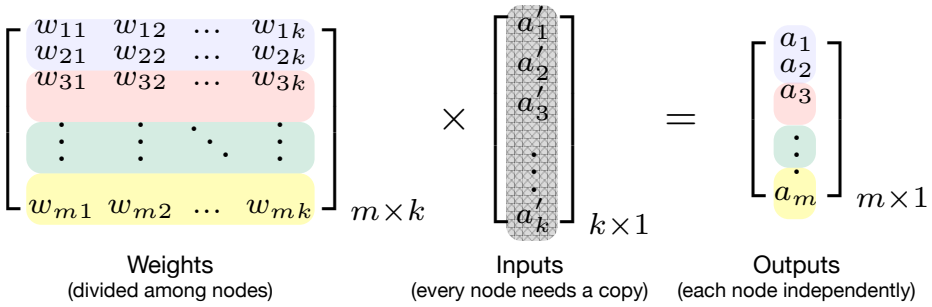


Figure 6.3: **Fully-Connected Layer Output Splitting and GEMM:** Distribution of output splitting for fully-connected layers.

6.2.1 Distribution and Matrix Operations

Fully-Connected Layer: A fully-connected layer performs Equation Equation 2.4 with GEMM. First we consider the matrix-matrix multiplication part, or $\mathbf{W^l a^{l-1}}$. Figure 6.3 illustrates how output splitting affects weight and output matrices for an example with four devices. Since each device calculates a set of separate outputs, the output matrix is created separately by each device (and concatenated later). Such separation in output generation also divides the weight matrix along the y-axis, which has a one-by-one relationship with the output matrix division. Each device needs a copy of the input matrix, and the input matrix is not divided.

In the input-splitting method, as Figure 6.4 depicts for the same four-devices example, the input matrix is divided between the devices. Similarly, the weights corresponding to those inputs are also divided along the x-axis among devices. Each device calculates partial sums for the entire output elements. Finally, all partial sums are aggregated to create the final output. Regarding bias and the activation function, we can extend the above reasoning. For output splitting, biases and the activation function can also be divided among the devices. But, for input splitting, both need to be applied after the aggregation. Since the majority of the computation time of DNNs is spent on matrix-matrix multiplications, such a difference does not have a big impact on computation time.

Convolution Layer: By utilizing Figure 2.3 for convolution layer, the channel-splitting

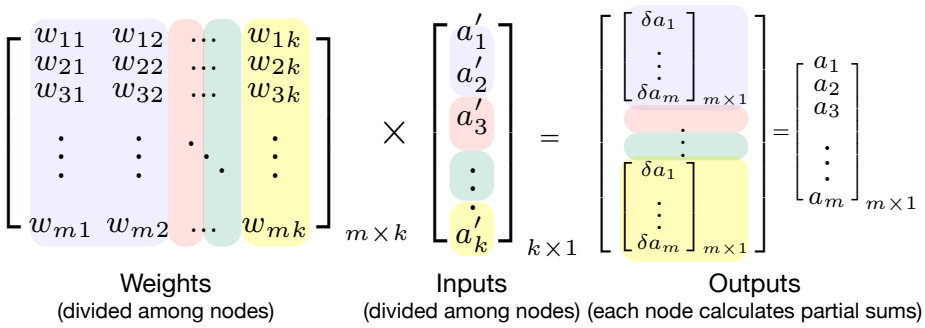


Figure 6.4: **Fully-Connected Layer Input Splitting and GEMM:** Distribution of input splitting for fully-connected layers.

method divides the filter weight matrix along the y-axis, as Figure 6.5 shows for two devices. Likewise, since the output is unrolled, such division translates to a similar along-the-y-axis division of the output matrix. Hence, channel splitting in convolution layers is the same as output splitting into fully-connected layers and any robustness analysis is applicable on both, but with a different set of weights and inputs (i.e., unrolled version of filters and patches in convolution layers).

In the spatial-splitting method, since each input patch is unrolled column-wise in the input matrix when we spatially divide the input, this division translates to an along-the-x-axis division of the input matrix. However, unlike input splitting in fully-connected layers, filter weights cannot be divided. Therefore, spatial splitting, as conceptually shown in Figure 6.6 for two devices, divides the input matrix of Equation 2.5 along the x-axis.

In the filter-splitting method, a close representation of input splitting for fully-connected layers, both filter weights and input are divided depth-wise. Since both filter weights and input are unrolled, we need to divide the weight and input matrices along the x- and y-axes, respectively. This distribution is similar to the outer product approach in matrix multiplication, versus the most commonly known algorithm of the inner product approach. Figure 6.7 shows this approach with two devices. Each device produces a partial sum for the entire elements. To create the final output, the final device needs to aggregate all the elements and apply the activation function.

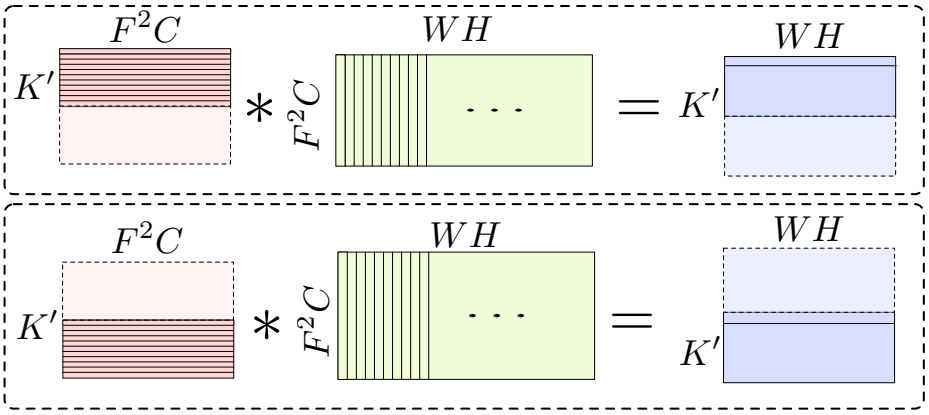


Figure 6.5: **Convolution Layer Channel Splitting and GEMM:** Distribution of channel splitting for convolution layers.

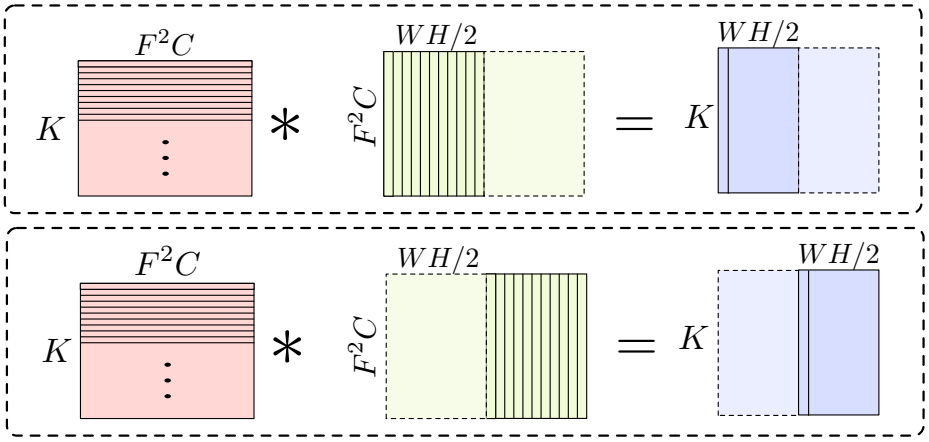


Figure 6.6: **Convolution Layer Spatial Splitting and GEMM:** Distribution of spatial splitting for convolution layers.

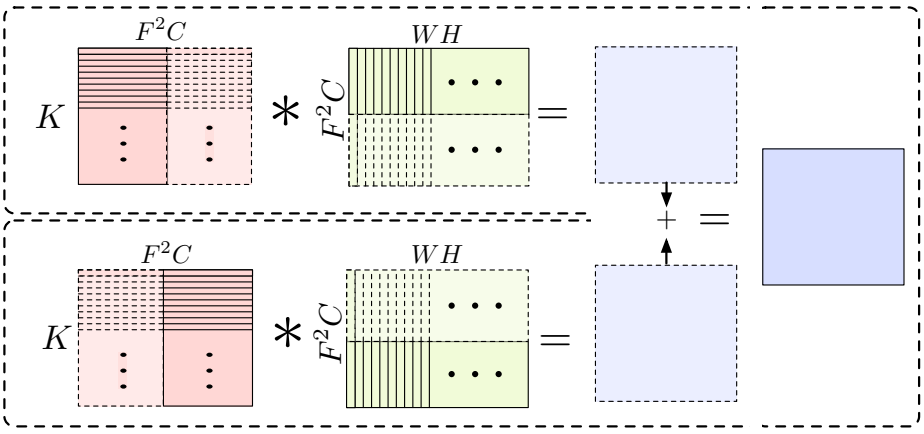


Figure 6.7: **Convolution Layer Filter Splitting and GEMM:** Distribution of filter splitting for convolution layers.

6.2.2 CDC Robustness with a Simple Example

We present a simple example of our CDC-based robustness to facilitate understanding. Consider a fully-connected layer with two input and output elements, written as:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}. \tag{6.1}$$

Assume that we perform output splitting. Now, by adding a row to the weight matrix with the value of $[w_{11} + w_{21} \;\; w_{12} + w_{22}]$, we can create the summation of two outputs, or $a_1 + a_2$. Therefore, with such addition, the above equation becomes:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{11} + w_{21} & w_{12} + w_{22} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_1 + a_2 \end{bmatrix}. \tag{6.2}$$

Since the summation of the weights can be done offline and is not dependent on inputs, we can rewrite about equation as:

$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w^{cdc}_{:1} & w^{cdc}_{:2} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a^{cdc} \end{bmatrix}. \tag{6.3}$$

The newly added weights to the weight matrix are the column sums of the weight matrix that is done offline before loading the weights. Now, with the addition of another device, we can guarantee to recover from one missing output with only a local subtraction in the final device. This method has three main benefits:

- First, this level of guarantee on all devices is just with an addition of one device, compared to a double modular redundancy method that duplicates all devices.
- Second, this method is faster than redoing all operations since the subtraction of two local values that we already have received is almost immediate than restarting all operations. This is because, the vanilla recovery method consists of loading a set of

new weights (corresponding to the missing values) in the final device, asking for the input from previous devices, and performing multiplications with all of its associated overhead of communication.

- Third, although we introduced the computations corresponding to a^{cdc}, these computations are similar in nature to the computations of a_1 and a_2. Hence, the distribution of these newly added computations follows the same rules and would not create an imbalance in the modified distribution.

6.2.3 Generalization of Robustness

In this section, we extend our simple scenario, in which each device computes only one output element, to a more realistic scenario, in which each device computes hundreds of elements. Similarly, we showcase the output-splitting method as our example. Assume a fully-connected layer performing the below equation:

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mk} \end{bmatrix}_{m\times k} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix}_{k\times 1} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}_{m\times 1} . \tag{6.4}$$

By distributing the computations among two devices, each of the devices perform the computations for $m/2$ of output elements. The computations per each device are

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\frac{m}{2}1} & w_{\frac{m}{2}3} & \dots & w_{\frac{m}{2}k} \end{bmatrix}_{\frac{m}{2}\times k} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{\frac{m}{2}} \end{bmatrix}, \text{and} \tag{6.5}$$

$$\begin{bmatrix} w_{(\frac{m}{2}+1)1} & w_{(\frac{m}{2}+1)2} & \dots & w_{(\frac{m}{2}+1)k} \\ w_{(\frac{m}{2}+2)1} & w_{(\frac{m}{2}+2)2} & \dots & w_{(\frac{m}{2}+2)k} \\ \vdots & \vdots & \vdots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mk} \end{bmatrix} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_k \end{bmatrix} = \begin{bmatrix} a_{(\frac{m}{2}+1)} \\ a_{(\frac{m}{2}+2)} \\ \vdots \\ a_m \end{bmatrix} \tag{6.6}$$

in which input matrices are the same, but the weight matrix is divided along the y-axis. Each device creates separate parts of the output matrix. To introduce robustness, the new weight matrix would be as follows:

$$\begin{bmatrix} w_{11} + w_{(\frac{m}{2}+1)1} & w_{12} + w_{(\frac{m}{2}+1)2} & \dots & w_{1k} + w_{(\frac{m}{2}+1)k} \\ w_{21} + w_{(\frac{m}{2}+2)1} & w_{22} + w_{(\frac{m}{2}+2)2} & \dots & w_{2k} + w_{(\frac{m}{2}+2)k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\frac{m}{2}1} + w_{m1} & w_{\frac{m}{2}2} + w_{m2} & \dots & w_{\frac{m}{2}k} + w_{mk} \end{bmatrix}_{\frac{m}{2}\times k} . \tag{6.7}$$

By multiplying this new weight matrix with inputs, the below output matrix is created:

$$\begin{bmatrix} a_1 + a_{(\frac{m}{2}+1)} \\ a_2 + a_{(\frac{m}{2}+2)} \\ \vdots \\ a_{\frac{m}{2}} + a_m \end{bmatrix}_{\frac{m}{2}\times 1} , \tag{6.8}$$

which is is the summation of two output matrices in Equation 6.5 and Equation 6.6. Therefore, by introducing such a weight matrix as Equation 6.7, we can introduce robustness. Similar to our simple example, the computation of this new weight is done offline, recovery has a close-to-zero latency, the robustness covers all devices, and the new computation does create an imbalanced distribution.

In contrast, splitting methods that work based on dividing the input matrix among the devices does not yield similar benefits. To illustrate why, we study input splitting among two devices for the computation of the same fully-connected layer presented in Equation 6.4. Input splitting for fully-connected layers divides the input and the weight matrix along the

x-axis. Accordingly, the computations per device are:

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1\frac{k}{2}} \\ w_{21} & w_{22} & \dots & w_{2\frac{k}{2}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{m\frac{k}{2}} \end{bmatrix}_{m\times\frac{k}{2}} \times \begin{bmatrix} a'_1 \\ a'_2 \\ \vdots \\ a'_{\frac{k}{2}} \end{bmatrix}_{\frac{k}{2}\times 1} = \begin{bmatrix} \delta a_1 \\ \delta a_2 \\ \vdots \\ \delta a_m \end{bmatrix}_{m\times 1} \tag{6.9}$$

$$\begin{bmatrix} w_{1(\frac{k}{2}+1)} & w_{1(\frac{k}{2}+2)} & \dots & w_{1k} \\ w_{2(\frac{k}{2}+1)} & w_{2(\frac{k}{2}+2)} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m(\frac{k}{2}+1)} & w_{m(\frac{k}{2}+2)} & \dots & w_{mk} \end{bmatrix}_{m\times\frac{k}{2}} \times \begin{bmatrix} a'_{\frac{k}{2}+1} \\ a'_{\frac{k}{2}+2} \\ \vdots \\ a'_k \end{bmatrix}_{\frac{k}{2}\times 1} = \begin{bmatrix} \delta a_1 \\ \delta a_2 \\ \vdots \\ \delta a_m \end{bmatrix}_{m\times 1} . \tag{6.10}$$

Each of the above equations calculates a partial sum. However, as seen, no share factor exists between the two computations. Therefore, if a third device wants to perform coded distribution, it needs to perform the entire calculations of Equation 6.9 and Equation 6.10. Such an approach creates unbalanced work between the devices, and has no advantage over just replicating the entire work as modular redundancy methods do.

Distribution Techniques Suitable for Robustness: For the distribution methods we introduced in Chapter 4, based on the aforementioned discussion, only some methods are suitable for our CDC-based robustness. Such suitable methods do not split the input elements but split the weights. Table 6.1 provides a summary of all the presented methods and whether they are suitable for robustness. For fully-connected layers, the output-splitting method is suitable for robustness. For convolution layers, the channel-splitting method has similar characteristics. Unfortunately, the rest of the distribution methods are not suitable for robustness. This is because to introduce robustness, these methods need to actually perform the entire computation again, which including the communication overhead. For instance, in spatial splitting, although every device has all the weights, they only own some part of the input. Therefore, with our technique, we need another device performing the computation

Table 6.1: Distribution Techniques Suitable for Robustness.

Layer	Model-Parallelism Method	Divides Input	Divides Weight	Divides Output	Suitable for Robustness
fc	Output	✗	✓	✓	Yes
	Input	✓	✓	✗	No
conv	Channel	✗	✓	✓	Yes
	Spatial	✓	✗	✓	No
	Filter	✓	✓	✓	No

based on the summation of the input parts. Since input elements change, computing such a summation has an overhead during the runtime (2x compute). The filter-splitting method also suffers from the fact that no element from the input or weights is shared between computing devices.

6.3 Experimental Studies

We evaluate our method on a distributed system with RPi [41] with 1.2 GHz Quad Core ARM Cortex-A53 CPU and a 900 MHz 1 GB RAM LPDDR2 memory. We choose RPi because they represent the de facto choice for several edge and edge use cases, they are readily available, and they allow common software packages. Our implementation is created with a software stack based on Docker containers. We use Keras 2.1 [124] with the TensorFlow backend (version 1.5) [117]. For RPC calls and serialization, we use Apache Avro [125]. A local WiFi network with the measured bandwidth of 94.1 Mbps and a measured client-to-client latency of 0.3 ms for 64 B is used.

Task Creation & Assignment: The policy of task creation in edge-based distributed DNN systems is done with either profiling or heuristics that use common monitoring/managing tools such as Kubernetes. The policies create tasks per device for a given DNN architecture by studying its memory footprint, computation requirement, and communication overhead. Regardless of the policy that finds the optimal distribution (out of scope of this paper, see [116]), all the pre-trained weights are loaded to each device storage so that a device can switch its assigned task easily if needed. For each number of available devices, a single task allocation file is loaded to all devices and each device performs its allocated tasks based on

the file. In our implementation, we use an IP table file to assign tasks to each RPi. CDC weights are also created offline and loaded to the storage. In the case of a failure, the system uses another pre-defined distribution file with fewer devices that has a lower performance. In such a case, since the detection of a missing device takes time, the system mishandles many requests. Our proposes solution that has tolerance to such failures, so the system never loses a request. Additionally, with a close-to-zero recovery latency, the system proactively is more tolerant to straggler nodes.

Weight Storage: Each Pi has an SD card storage, for storing the weights, which is relatively inexpensive compared to the main memory. All trained weights are loaded to each Pi's storage (16 GB storage in our system), so each Pi can be assigned to execute any part of a layer. If local storage is limited, the assigned weight can also be shared on the network from a network-storage filesystem. This approach makes a tradeoff between how fast the switching time between different models can be and per-device storage usage. Additionally, note that the distribution method does not replace other methods, such as offloading to servers. The decision is the per-case basis and depends on several system-level decisions. The distribution offers the additional option of processing data locally.

6.3.1 System Recovery Case Studies

Case Study I: To depict the impact of how failures affect a system, we deploy AlexNet [1] on two edge systems. The first system, shown in Figure 6.8a, contains five devices. The first fully-connected layer is split with the output-splitting method between two devices with no robustness method. The black bars in Figure 6.9 show the latency of the system when performing single-batch inferences. Now, if device C experiences failure, as shown in Figure 6.8b, other devices need to perform the task assigned to the failed device. Since the task of device C is the computation of half of the first fully-connected layer, device D needs to perform this extra task in addition to its task. After the failure is detected, which takes tens of seconds, the red bars in Figure 6.9 depict the new shifted latency histogram of the

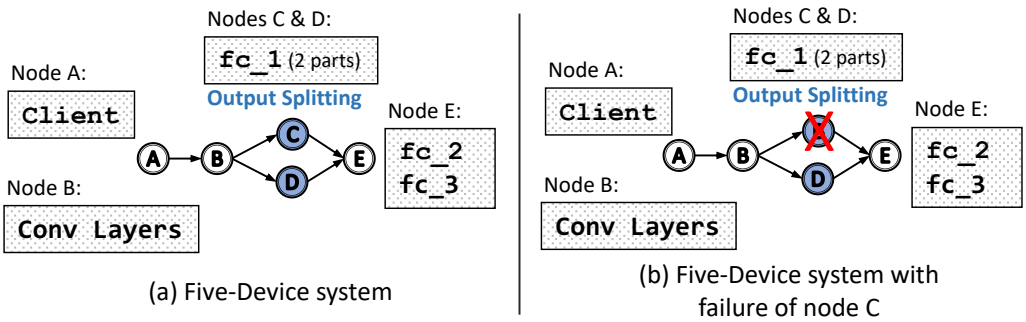


Figure 6.8: **Case study I:** AlexNet on a five-device system.

system. Based on our measurements, on average, the system experiences 2.4x slowdown after recovery. The system is not performing beneficial work during failure detection, and experiences significant slowdown afterward. However, with our method, the system does not experience any slowdown or service interruption.

Case Study II: As a remedy to failures, we deploy AlexNet on a six-device system. Figure 6.10a shows this system, in which an extra device is added for robustness using CDC. Note that our goal is to create robustness only for the first fully-connected layer and the extra device provides robustness to all the computations done on device D and E. If we experience failure, as Figure 6.10b shows, the performance of the system does not change. Additionally, during the operation without failure, we use the extra device to mitigate the straggler problem. Figure 6.11 and Figure 6.12 show the system latency with and without this mitigation, respectively. As shown, the range and the distribution of latencies are improved towards a better performance. Thus, in addition to robustness, we can exploit the

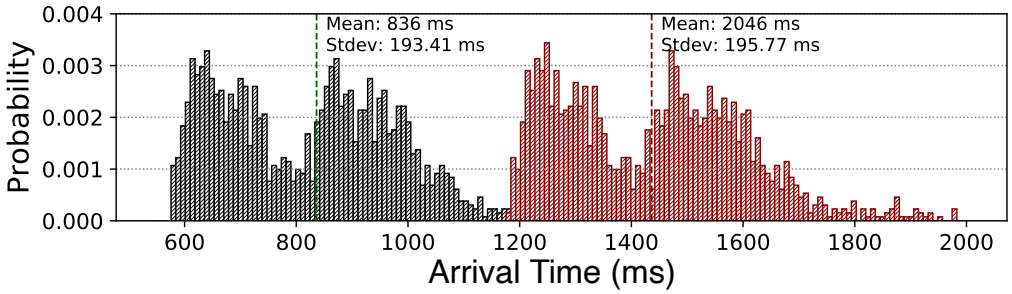


Figure 6.9: **Case study I:** Recovery latency with & without CDC.

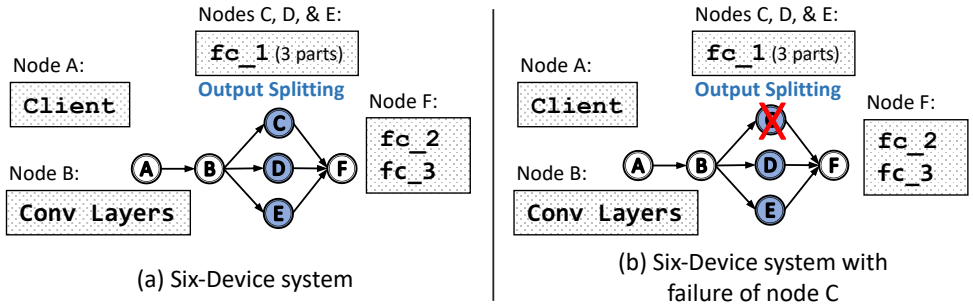


Figure 6.10: **Case Study II:** AlexNet on a six-device system.

extra device to increase the performance.

6.3.2 Straggler Mitigation

We study straggler mitigation benefits by extending the previous system. To initiate recovery, a device waits for a particular amount of time. By adjusting this waiting threshold in a device, we can treat our method as a solution for the straggler problem after receiving the necessary amount of data. A lower threshold reduces latency and thus increases performance. Straggler problem is more prominent with more devices, so we set up an experiment as Figure 6.13a shows for a system with four devices, each of which performs a split in a fully-connected layer. Figure 6.13b shows performance improvement of straggler mitigation with a diffident number of devices in a system. The performance improvement is compared with the same system, with the same number of devices, with no straggler mitigation. As seen, for more devices, straggler mitigation has better performance (up to 35%) compared

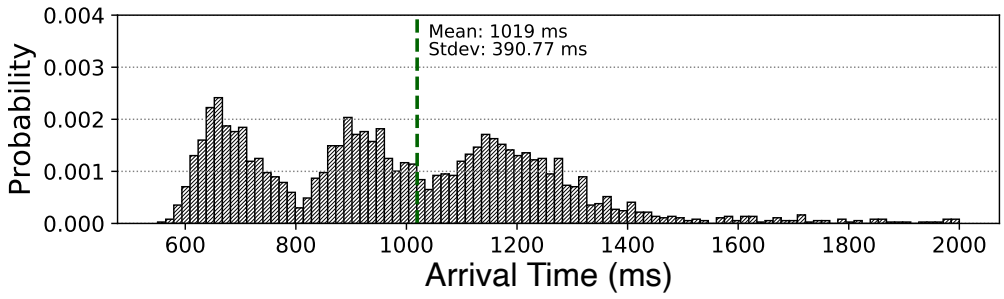


Figure 6.11: **Straggler Problem:** AlexNet latency histogram without straggler mitigation.

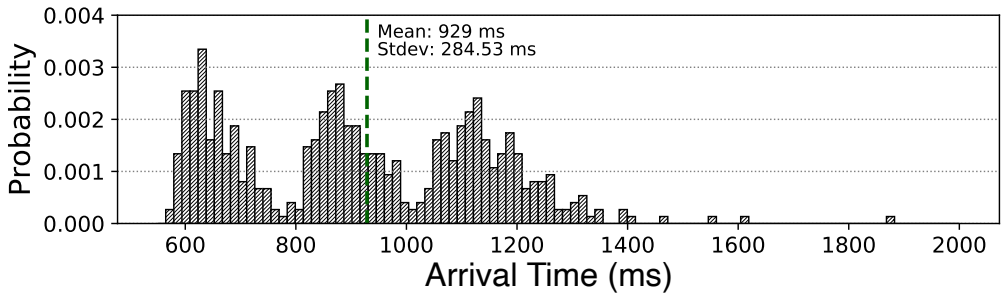


Figure 6.12: **Straggler Mitigation:** AlexNet latency histogram with straggler mitigation.

with a no-straggler-mitigation system with the same number of devices.

6.3.3 Full Model Coverage

In the system shown in Figure 6.10, devices with model parallelism are robust with CDC. For other devices, by the replication of the device's task (N-modular redundancy with $N = 2$, or 2MR), we can tolerate one failure in the entire system. Such a hybrid approach (CDC+2MR) could cover the entire system for failures. In a nutshell, our method covers any number of devices in one layer with just one additional device (this is for robustness to one failure). But, 2MR needs an additional device for each device. Therefore, our method has a *constant* cost with an additional number of devices; whereas 2MR requires a *linear* number of additional devices. In Figure 6.15, we study several DNNs [4, 9, 40, 37] with distributed implementations with tolerance to one failure with 2MR-only and CDC+2MR. Since CDC requires fewer devices than 2MR to cover the devices with model parallelism, the number of additional devices for full coverage for CDC+2MR is smaller than that of 2MR. The

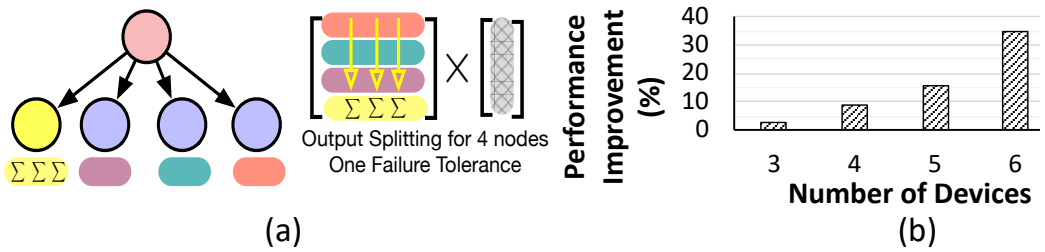


Figure 6.13: **Straggler Mitigation Study:** (a) a system setup for four devices. (b) Straggler mitigation performance.

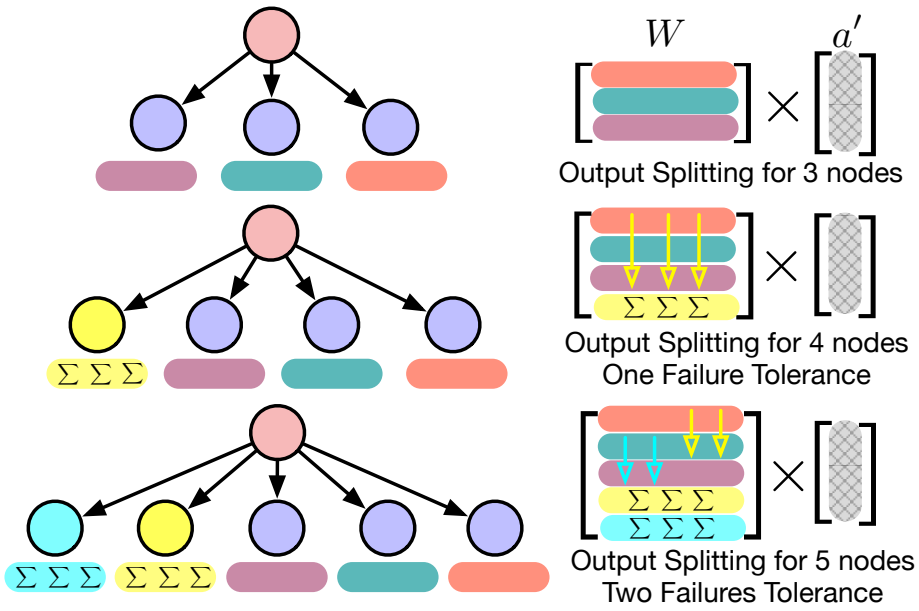


Figure 6.14: Tolerating multiple failures.

amount of difference depends on how many layers are distributed with model parallelism and how many devices are used per layer. For instance, Figure 6.15c and d depict two C3D distributions with different numbers of the devices for the layers that use model parallelism (two vs. three devices). We see that in Figure 6.15c and Figure 6.15d, with two additional devices, CDC+2MR, compared with 2MR with 44% and 36%, reaches the coverage of 67% and 73%. This is because C3D distribution has two layers with model parallelism. Therefore, compared with 2MR, CDC+2MR achieves better coverage. In summary, if we use model-parallelism for a layer with N number of devices, with $(1 + \frac{1}{N})$ times hardware cost, we can hide a single node's failure as opposed to 2x hardware cost in 2MR.

6.3.4 Discussions

The Introduced Computation: The introduced new computations for our CDC-based method are similar to that of underlying GEMM computations. This is because we add new wights to the weight matrix (or a variant of it). These new weights can be calculated without the user's input and at a library level. Therefore, there are no additional costs for reprogramming the applications. Moreover, since the nature of the computations for these

new weights is similar to that of DNNs, there is no need to design new kernels or distribution methods.

Extending Robustness To More Failures: Our discussions were focused on tolerating up to one failure. However, Extending to more than one failure is possible by adding new devices that perform computations based on the summation of some rows of weights instead of all of them. Figure 6.14 illustrates three setups in order of increasing tolerance to failures. The last setup tolerates two failures because new devices perform partial sums on the weights.[1] Thus, by utilizing idle devices with an overlapping set of weights, the robustness of the system increases.

6.4 Summary

By utilizing CDC, we proposed a method to introduce tolerance for the single-batch inferencing of DNNs. Single-batch inferencing is important in edge and near-the-edge computing domains because of the time-sensitivity of applications and the limited number of the requests in these domains. Our method exploits model-parallelism methods in prevalent DNN layers to add balanced computation for robustness. Model-Parallelism methods help us in achieving efficient system distribution by splitting the computation of single-batch inferencing among several edge devices. We studied model-parallelism methods and their underlying computation when being distributed. To this end, we extended CDC to provide a trade-off between computations and robustness on distributed edge-based systems.

[1] Note that the coverage to two failures is almost complete (partial error correction). We need Hamming-style coverage for full error correction.

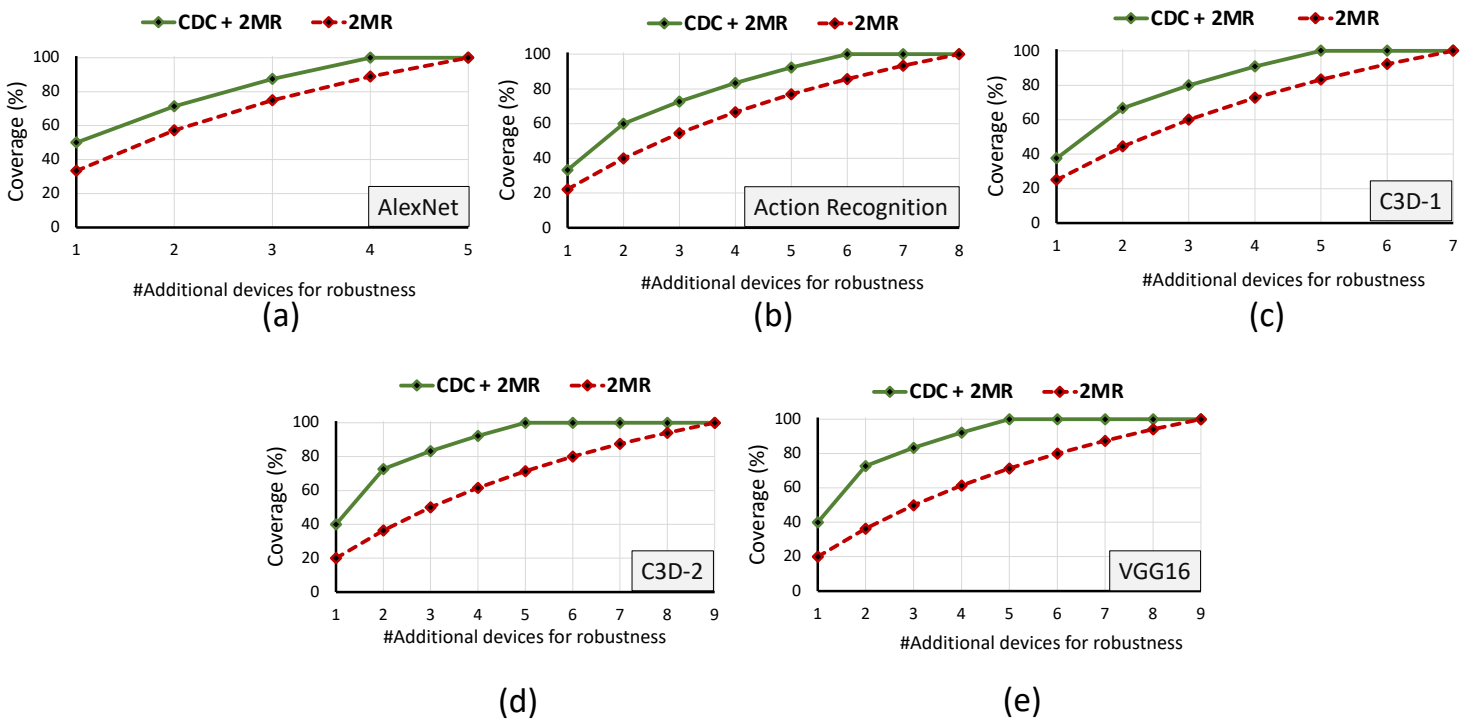


Figure 6.15: **Model Coverage Studies:** (a) AlexNet, (b) action recognition models studied in section 4.5, (c) C3D first distribution, (d) C3D second distribution, and (e) VGG16

CHAPTER 7
CONCLUSION & DISCUSSIONS

7.1 Conclusion

Edge systems constitute an increasingly vital portion of computing systems to which we entrust large portions of our daily lives. These systems are broadly defined as anything that is not considered large-scale datacenter machines. Edge systems are everywhere, from Internet of Things (IoT) devices, the number of which has already surpassed the world's population [169], to autonomous vehicles and robots, which are increasingly gaining ground in transportation and automation, projected to be valued at more than $1.1 trillion by 2026 [170]. Smartphones, with a market share of $2.3 trillion by 2025, are also edge systems. A common feature in future edge systems is their intelligence in adapting to various circumstances. Unlike traditional computing systems that are engineered with abundant resources and are under continuous monitoring, intelligent edge systems must operate within design constraints that are quite different [171]. Any intelligent edge system must arbitrate between its limited resources while guaranteeing correct execution under uncertainty. Despite advances in this area, current solutions are a mishmash of borrowed ideas and tools, which restrain achieving the full capabilities of edge systems and maximizing their performance. The ultimate goal of my research is to enable efficient and effective edge systems to extend their reach by exploiting the hardware-software synergy and targeting optimal points within their unique multidimensional design-space.

The challenge of today's edge systems limiting their capabilities stems from the fact that edge systems must operate in real-world conditions by considering various factors, such as monetary cost, power consumption, real timeliness, weight, safety, privacy, and connectivity. Furthermore, despite huge technological advances in this area, usually problems have been

approached in isolation and the end-to-end design-space tradeoffs are largely unknown. Realizing intelligent edge systems requires a tight integration with autonomous features. And to realize such autonomy, these systems must handle large quantities of raw data that must be processed and analyzed in real time, often with deep neural networks. However, inadequate power and computing resources restrict full-scale autonomy in these systems. Finally, ease of programmability and generality accompanied by optimized and fast execution adds another layer to this challenge. These factors limit the widespread applicability of edge systems and hinder their advancement. The current unique characteristics of edge systems and our excitement over the current possibilities, such as autonomous vehicles, are evidence of how immensely impactful future intelligent edge systems could be in our society.

The main portion of my research during my Ph.D., presented in this dissertation [172, 149, 115, 116, 173, 16, 174, 175, 128, 176, 166, 177, 130, 131], focused on performing DNN inference locally (in the edge environments) with efficient and reliable distribution with the additional help of parallelization. In fact, during my graduate study, I witnessed the advancements of deep neural networks (DNNs) making revolutionary changes in edge domains such as robotics [10, 11, 178, 115, 179], unmanned aerial vehicles (UAVs) [12, 13, 180, 181, 182], and Internet-of-things (IoT) [58, 14, 59, 183, 116, 184]. In these domains, such as smart homes/cities/offices (*e.g.*, connected cameras, gaming consoles, TVs, routers) or collaborative robots/drones (*e.g.*, disaster relief [185, 186, 187], agriculture [188, 189], mining [190], construction [191], mapping [187, 192]), (i) ensuring an acceptable accuracy is enough (*e.g.*, detecting human sound in a disaster area with either 87% or 90% accuracy necessitates more investigation); (ii) the network of devices is standalone (*i.e.*, Internet connection is not available/necessary); and (iii) the network has a unified ownership and hence communication among devices is not hazardous (*e.g.*, IoTs at home, robots at warehouse). In such domains, executing inference in-the-edge could enable several features; however, performing the heavy inference computations locally is still. The proposed techniques in this dissertation are further summarized as follows:

- Chapter 4 presented various model-parallelism methods for distributing and parallelizing the computation of single-batch inferences. The focus was on convolution and fully-connected layers that dominate CNNs and LSTMs runtime [87]. Our methods, input, and output splitting in fully-connected layers and spatial, channel, and filter splitting in convolution layers, were specifically designed so that they are applicable on top of any high-level frameworks such as TensorFlow [117] or PyTorch [164] (check subsection 4.3.4 for other types of methods). Later in the chapter, we explained how these methods are expandable to video streams to be used in video action recognition models. This chapter also proposed two work distribution methods: Offline with the help of profiling, and online with the help of monitoring. Finally, the chapter concludes with experiments on real implemented systems of up to 12 Raspberry Pi 3s [115, 116, 16].

- Chapter 5 went one step further than the presented distribution methods for current DNN models. This chapter focused on customizing and introducing new DNN models that are tailored for efficient distribution. This is because, with the current model architectures, model parallelism methods cannot reduce communication, memory usage, and computations at the same time (see Table 5.1). In section 5.1, we first proposed a handcrafted solution in customizing DNN models with low-communication parallelization (LCP) method while also presented and accompanying special hardware design for low-power edge devices. LCP method reduced communication by replacing a single, wide, and deep model with several narrow ones that only communicate for input and pre-final activations. LCP also allowed inter-layer parallelism due to the single-chain dependency between consecutive layers. Second, in section 5.2, we endeavored to inspire discussion for new concurrent architectures by addressing the single-chain data dependency in current architecture designs. This section improved the LCP method with a neural architecture search (NAS) method that encapsulated the important metrics such as communication, load balancing, and overlapped com-

putations, by reformulating the problem as a hypergraph partitioning problem [130, 131].

- In the last chapter, Chapter 6, we increased the reliability of the deployed distributed system using a new recovery method based on coded distributed computing (CDC). To do so, we analyzed how the methods presented in Chapter 4 change the underlying matrix-matrix computations of the models. Then, we introduced an extra coded computation per device that are similar in nature to those of DNNs. Thus, it was easier to balance the introduced work among edge devices and reduce the deployment cost. Additionally, since our method was implemented in the underlying matrix-matrix computations of DNN layers, it did not require extensive changes to the user's program and is implemented at the library level. Moreover, our method, even at the time of failures, provided close-to-zero recovery time, which is necessary for critical time-sensitive tasks. Finally, compared with conventional modular redundancy methods that introduce redundancy in computation by introducing a linear number of additional devices, our method had a constant cost with the increasing number of devices. We demonstrated our method on distributed systems comprising of Raspberry Pis.

To realize intelligent edge systems, my future research focuses on developing software ecosystems and hardware designs by pushing the frontiers of current technologies. I investigate efficient hardware-software co-designs, algorithmic modifications, and bit-level operations to enable efficient, secure, reliable, and cost-efficient edge systems. By discovering the unique characteristics of design-space tradeoffs in edge systems and their applications, I seek to overcome the aforementioned obstacles to make edge systems intelligent by envisioning the type of algorithms that are executed and examining critical end-to-end system tradeoffs, such as latency. Compared with current time-consuming and costly approaches in designing ad-hoc edge systems for each individual use case, I am wishful that my research can extract generality and provide solutions to a wide variety of edge systems.

7.2 Discussions

This section discusses assumptions made during the studies, how they would affect the results, and the practical approaches to fix them. Next, I discuss *my own opinion* on tools, industry trends, the gap in the current technologies, and general artificial intelligence.

7.2.1 Assumptions in The Studies

Most of the results in this thesis were measured using real implemented systems. We reached this stage, compared to only simulation results, by assuming some facts and using several open-sourced tools. The following summarizes the assumptions and presents discussions on other options.

- *Centralized Communication:* From a network implementation perspective, all the systems assume a central router routing all the packets among nodes. This mode of communication has a direct contrast with pair-to-pair (P2P) communications, in which each node can directly talk to other nodes. Using centralized communication like using a WiFi router has certain implementation benefits. For instance, it is much easier to implement, acquire equipment, and tools with a WiFi router than using P2P on an edge node. From a performance perspective, however, a centralized communication protocol may not give the best latency. In centralized communication, each packet must travel from the source to the router, and then to the destination. Thus, the minimum hop count is two. In contrast, in P2P the minimum hop count is one. But, this is not always true since even in P2P communication, some packets need to travel across several nodes to reach the destination. In practice, the real hop count depends on the topology of the network, P2P protocol used (*e.g.*, routing and discovery method), and the type of communication protocol support by the edge devices (*e.g.*, Raspberry Pi can use WiFi-based P2P, but not all edge devices can). Moreover, most P2P protocols do not support high bandwidth communications (and

even if available they will not be supported by most edge devices). Finally, using P2P communication would certainly increase the power consumption of the edge devices compare to using centralized communication due to the cost of additional time and packets for re-routing and discovery operations. Thus, it is difficult to say if using P2P leads to better or worst performance metrics. In the thesis, the latency of one round trip with our routers and Raspberry Pi 3Bs are in orders of 100-200 ms. This latency is not that slow or the shortest possible latency with WiFi technology. Therefore, I believed the experiment results are accurate for most communication protocols. But the aspect of communication protocol and their performance effect certainly requires a much deeper study that is out of the scope of this dissertation.

- *Computations on Network Router:* In our experiments, the router itself does not collaborate in performing computations. The collaboration is in fact possible since routers have similar processing capabilities as a single Raspberry Pi in our experiments. But stating that all the computations can be done on the router is not practical. Moreover, in-router computations are considered to be part of the *fog computation paradigm*, which is not the focus of edge computing. Furthermore, the existence of a router does not necessarily mean that the router is also a modem and is connected to the Internet.

- *Sharing one Input Image:* All the designs assume that a sensor (with limited capabilities) or a camera broadcasts one image to all devices that are performing the first layer (either in current models with model parallelism or new models). The input resolution is usually a 224x224x3 image, the transmit time is not long. However, if the input size for a model is large, another option is to process the first few layers on the first device. The process (*i.e.*, compression) helps to reach smaller intermediate features that take shorter to transmit. If even more compression is needed, we can develop a bottleneck layer (*i.e.*, encoder and decoder) to distill the information before transmitting it.

- *Using Sensors While Performing Tasks:* We used sensors and edge devices while they are idle in a network. If a given device is currently busy, we reallocate the jobs to other devices or use a fewer number of devices in our distribution. As discuss in Chapter 4, all our distribution methods have included this scenario in their assumptions.

- *System-Level Supporting Programs:* In research-oriented projects, it is customary to ignore system-level supporting programs since they have a small performance impact on the final results and the details of the exact use case are vague. However, to realize systems developed in this research, we must employ several system-level supporting programs that are as important as inference computations. For instance, we must make sure the images are not blurry before performing the inference, or the inference result passes the minimum accuracy metrics to be reported. These problems are extremely interesting, however, their exact details depend on our use case.

- *How Models Affects Our Methods:* Usually, model designers are oblivious to the performance of their newly invented models as long as the models hit the highest accuracy. Nevertheless, as the industry is moving to implement these models in a large scale from datacenters to user devices, they have realized such isolation of accuracy and performance mostly does not produce a practical outcome. This isolation has come in various forms. From offloading all the optimizations to hardware and product teams to actually help the teams to find better models. As a result, the current state of approach to design state-of-the-art *practical* models includes experts from/in both areas. As a result, we are seeing a surge of a new type of layers with much better characteristics in performance on real systems (*e.g.*, Fire modules in SqueezeNet [193] in or EfficientNet [194]). In fact, ResNeXt [195] has grouped convolution layers that follow a similar philosophy as the LCP model designs. Therefore, to distribute ResNeXt, we can execute each of the branches in the group convolution layer on a node independently. Still, however, we must figure out how to distribute early and

final layers that are not grouped convolution. In my view, this trend of new models that include performance as one of their goals would be much successful in the future than models that only target ultimate accuracy to make it to the news headlines.

- *Current Status of Edge-Computing Industry and Tools:* My research focused on distribution of deep neural network computations in the edge. Edge-computing startups are looking to implement these models on a *single node* with customized accelerators and even microcontrollers. The models used in such designs are lightweight and thus does not provide same accuracy metrics of current state-of-the-art models. Nevertheless, I believe such endeavors are the first step to realize distribution. From a practical standpoint, we must first get the most performance from the single node (*e.g.*, similar to single-core CPUs in 2006), see a diminishing return, and then start utilizing the benefits of distribution. From a research standpoint, however, we must understand how and why distribution would help before reaching to the point where using distribution is inevitable. This is because, using this research, we can then develop models and systems that also are considering the fact that one day, we might use distribution. Furthermore, to fill the gap in current tools, we must know what is missing that research in this direction can reveal.

- *Using Proposed Methods for Datacenter-Based Inference:* Re-purposing the methods presented in this dissertation to be used in datacenter-based inference is possible in a high level of abstraction. To do so, we must redefine the cost of distribution and create larger divisions per model. Given that models being executed in datacenters are usually larger (*e.g.*, large input resolution), creating larger divisions is possible. On the other hand, redefining the cost of distribution at the datacenter level requires more throughout the study. Additionally, some datacenters have dedicated customized hardware units for inference computations. These novel units enable extra methods for model parallelism that is not available on edge devices. This is because similar

to DNN accelerators, these units have defined and implemented new DNN dataflows for execution. Kwon et al. [118] has done great work in distilling these dataflows. In summary, although the presented methods can be used in datacenter-based inference at a high level, we must account for several extra effects that are only applicable for datacenters.

- *General Artificial Intelligence and Consciousness:* I want to dedicate this last paragraph to my view on general artificial intelligence (AGI), consciousness, and research responsibility. When I started to learn about deep learning, the consensus was that deep learning and neural networks would pave the way for AGI and consciousness, the next big (and maybe final) thing for humans to achieve. Although deep learning has historical roots going back decades, the breakthroughs in vision revived the field. Nevertheless, now that I am writing this, the goal of achieving AGI is more elusive than before (at least for me). Neural networks and deep learning have almost no relation to how our mind works and why we are conscious. Only imitating our brain structure on the surface in deep learning, with several assumptions, seems to have led us to wander. Our designed artificial intelligence has limited scope and cannot be generalized to the simplest of tasks humans do, even with billions of operations. Gary Marcus [196], a critic of deep learning, would provide more reasons if you are still a believer. Even if deep learning would lead to AGI, we are still way behind in our technology to make a 40 W brain-like computer that will operate independently.

  This leads me to discuss consciousness. I have followed consciousness research since I have started the research on deep learning (not sure if calling it deep learning is appropriate anymore). I believe for realizing AGI, we must have some form of consciousness. This is why we cannot build a generalized model since our knowledge of consciousness is limited and we continue to be ignorant about it. We simply disregard conscious beings around us while trying to build our fancy supercomputers with huge computational capabilities that *have and will have* zero consciousness. The

problem is that we may never be able to build a conscious agent. This is because as Thomas Nagel says, “And if there is conscious life elsewhere in [the] universe, it is likely that some of it will not be describable even in the most general experiential terms available to us... [However,] this does not prevent us each from believing that the other’s experience has such a subjective character.” [197]. So, it would be impossible for us to conceive what is like to be another conscious being even if we end up having one. However, this does not prevent us from acknowledging other’s experiences and therefore their feelings. This brings us to my last topic, research responsibility. As researchers, we are responsible for the impact of our research. Although deep learning fails to capture consciousness, still, it provides a powerful tool to process all kinds of data. With our exponential progress, most researchers fail to see the big picture and the impacts they are making on our planet and society. This will not get easier in the future as researchers would scramble for the next big thing. And, if in the future, we build a conscious agent, the burden is significant, if we realize it.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *26th Annual Conference on Neural Information Processing Systems (NIPS)*, ACM, 2012, pp. 1097–1105.

[2] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *ICML'8*, ACM, 2008, pp. 160–167.

[3] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *ICLR'15)*, ACM, 2015.

[4] M. S. Ryoo, K. Kim, and H. J. Yang, "Extreme low resolution activity recognition with multi-siamese embedding learning," in *AAAI'18*, IEEE, 2018.

[5] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *NIPS'14*, ACM, 2014, pp. 568–576.

[6] S. Zhang, L. Yao, A. Sun, and Y. Tay, "Deep learning based recommender system: A survey and new perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, p. 5, 2019.

[7] E. Ohn-Bar and M. M. Trivedi, "Looking at humans in the age of self-driving and highly automated vehicles," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 1, pp. 90–104, 2016.

[8] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[9] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters (RA-L), Invited to IEEE/RSJ International Conference on Intelligent Robots and Systems 2018 (IROS)*, vol. 3, no. 4, pp. 3709–3716, 2018.

[10] A. Giusti, J. Guzzi, D. C. Cireşan, F.-L. He, J. P. Rodríguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, *et al.*, "A machine learning approach to visual perception of forest trails for mobile robots," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 661–667, 2016.

[11] M. Pfeiffer, M. Schaeuble, J. Nieto, R. Siegwart, and C. Cadena, "From perception to decision: A data-driven approach to end-to-end motion planning for autonomous

ground robots," in *2017 ieee international conference on robotics and automation (icra)*, IEEE, 2017, pp. 1527–1533.

[12] A. Singh, B. Ganapathysubramanian, A. K. Singh, and S. Sarkar, "Machine learning for high-throughput stress phenotyping in plants," *Trends in plant science*, vol. 21, no. 2, pp. 110–124, 2016.

[13] H. Lu, Y. Li, S. Mu, D. Wang, H. Kim, and S. Serikawa, "Motor anomaly detection for unmanned aerial vehicles using reinforcement learning," *IEEE internet of things journal*, vol. 5, no. 4, pp. 2315–2322, 2018.

[14] O. B. Sezer, E. Dogdu, and A. M. Ozbayoglu, "Context-aware computing, learning, and big data in internet of things: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 1–27, 2018.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations*, ACM, 2016.

[16] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *Proceedings of IEEE International Symposium on Workload Characterization*, 2019.

[17] S. Li, L. Da Xu, and S. Zhao, "The internet of things: A survey," *Information Systems Frontiers*, vol. 17, no. 2, pp. 243–259, 2015.

[18] F Biscotti, J Skorupa, R Contu, *et al.*, "The impact of the internet of things on data centers," *Gartner Research*, vol. 18, 2014.

[19] I. Lee and K. Lee, "The internet of things (iot): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[20] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: The internet of things architecture, possible applications and key challenges," in *FIT'12*, IEEE, 2012, pp. 257–260.

[21] Y. Wang, H. Li, and X. Li, "Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices," in *ICCAD'16*, 2016, pp. 1–6.

[22] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *ICLR'16*, ACM, 2016.

[23] B. McDanel, S. Teerapittayanon, and H. Kung, "Embedded binarized neural networks," in *EWSN'17*, 2017, pp. 168–173.

[24] S. Bang, J. Wang, Z. Li, C. Gao, Y. Kim, Q. Dong, Y.-P. Chen, L. Fick, X. Sun, R. Dreslinski, *et al.*, "14.7 a 288uw programmable deep-learning processor with 270kb on-chip weight storage using non-uniform memory hierarchy for mobile intelligence," in *ISSCC'17*, IEEE, 2017, pp. 250–251.

[25] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," in *ISCA'16*, ACM, 2016, pp. 255–266.

[26] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2017, pp. 615–629.

[27] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge, "A hybrid approach to offloading mobile image classification," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2014*, IEEE, 2014, pp. 8375–8379.

[28] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, IEEE, 2017, pp. 328–339.

[29] Microsoft, *Embedded learning library (ell)*, https://microsoft.github.io/ELL/, [Online; accessed 4/4/21], 2017.

[30] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV'16*, Springer, 2016, pp. 525–542.

[31] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[32] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An execution framework for deep neural networks on resource-constrained devices," in *MobiSys'16*, 2016.

[33] Facebook, *Caffe2go: Delivering real-time ai in the palm of your hand*, https://code.facebook.com/posts/196146247499076/delivering-real-time-ai-in-the-palm-of-your-hand/, [Online; accessed 4/10/21], 2017.

[34] Google, *Introduction to tensorflow lite*, https://www.tensorflow.org/mobile/tflite/, [Online; accessed 4/10/21], 2017.

[35] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[36] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16*, IEEE, 2016, pp. 770–778.

[37] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*, ACM, 2015.

[38] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR'15*, IEEE, 2015, pp. 1–9.

[39] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *arXiv preprint*, 2016.

[40] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning Spatiotemporal Features with 3D Convolutional Networks," in *Computer Vision (ICCV), 2015 IEEE International Conference on*, IEEE, 2015, pp. 4489–4497.

[41] R. P. Foundation, *Raspberry pi 3b+*, www.raspberrypi.org/products/raspberry-pi-3-model-b/, [Online; accessed 4/4/21], 2017.

[42] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[43] Y. LeCun, "The mnist database of handwritten digits," *http://yann.lecun.com/exdb/mnist/*, 1998.

[44] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[45] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, 2008.

[46] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[47] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a

tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 1–12.

[48] P. Mattson, V. J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, *et al.*, "Mlperf: An industry standard benchmark suite for machine learning performance," *IEEE Micro*, vol. 40, no. 2, pp. 8–16, 2020.

[49] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *ISCA'17*, ACM, 2017, pp. 13–26.

[50] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "Vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Press, 2016, p. 18.

[51] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "Cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[52] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe con troll: Shallow ideas to speed up deep learning," in *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, ACM, 2015, p. 2.

[53] B. Kågström, P. Ling, and C. Van Loan, "Gemm-based level 3 blas: High-performance model implementations and performance evaluation benchmark," *ACM Transactions on Mathematical Software (TOMS)*, vol. 24, no. 3, pp. 268–302, 1998.

[54] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[55] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[56] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[57] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[58] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, L. Su, and T. Abdelzaher, "Deep learning for the internet of things," *Computer*, vol. 51, no. 5, pp. 32–41, 2018.

[59] H. Li, K. Ota, and M. Dong, "Learning iot in edge: Deep learning for the internet of things with edge computing," *IEEE Network*, vol. 32, no. 1, pp. 96–101, 2018.

[60] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[61] Ofer Dekel - Microsoft Research, "Compiling ai for the edge," *SysML 2019 Keynote*, 2019.

[62] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[63] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proceeding Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, ACM, 2011, p. 4.

[64] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[65] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *NIPS'12*, ACM, 2012, pp. 1223–1231.

[66] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[67] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *44th International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 548–560.

[68] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 2181–2191.

[69] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[70] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[71] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Eridanus: Efficiently running inference of dnns using systolic arrays," *IEEE Micro*, 2019.

[72] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplication," *arXiv preprint arXiv:1412.7024*, 2014.

[73] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 1742–1752.

[74] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[75] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or- 1," *arXiv preprint arXiv:1602.02830*, 2016.

[76] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, automation and Test in eurpe (Date)*, IEEE, 2017, pp. 1396–1401.

[77] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6848–6856.

[78] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger, "Condensenet: An efficient densenet using learned group convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2752–2761.

[79] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[80] J. Kim, Y. Park, G. Kim, and S. J. Hwang, "Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, JMLR. org, 2017, pp. 1866–1874.

[81] Intel Corp., *Intel movidius neural compute sdk*, https://movidius.github.io/ncsdk/tools/tools_overview.html, [Online; accessed 4/5/21], 2019.

[82] Intel Corp., *Intel movidius neural compute sdk*, https://software.intel.com/en-us/movidius-ncs, [Online; accessed 4/5/21], 2019.

[83] Nvidia Corp., *Nvidia tensorrt*, developer.nvidia.com/tensorrt, [Online; accessed 4/27/21].

[84] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: End-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, pp. 1–15, 2018.

[85] J. Dean, *Machine learning for systems and systems for machine learning*, 2017.

[86] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 75–84.

[87] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[88] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 92–104.

[89] S. Wang, D. Zhou, X. Han, and T. Yoshimura, "Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 1032–1037.

[90] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 609–622.

[91] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Press, 2016, p. 17.

[92] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 161–170.

[93] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2016, pp. 16–25.

[94] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2016, pp. 26–35.

[95] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 243–254.

[96] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 267–278.

[97] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, ACM, vol. 49, 2014, pp. 269–284.

[98] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ACM, 2019, p. 233.

[99] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[100] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016.

[101] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[102] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[103] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

[104] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "Mnasnet: Platform-Aware Neural Architecture Search for Mobile," *arXiv preprint arXiv:1807.11626*, 2018.

[105] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," *arXiv preprint arXiv:1904.01569*, 2019.

[106] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "A fundamental tradeoff between computation and communication in distributed computing," *IEEE Transactions on Information Theory*, vol. 64, no. 1, pp. 109–128, 2018.

[107] N. S. Ferdinand and S. C. Draper, "Anytime coding for distributed computation," in *Communication, Control, and Computing (Allerton), 2016 54th Annual Allerton Conference on*, IEEE, 2016, pp. 954–960.

[108] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," in *Advances In Neural Information Processing Systems*, 2016, pp. 2100–2108.

[109] R. Bitar, P. Parag, and S. El Rouayheb, "Minimizing latency for secure distributed computing," in *Information Theory (ISIT), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 2900–2904.

[110] K. Lee, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Coded computation for multicore setups," in *Information Theory (ISIT), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 2413–2417.

[111] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," in *ACM SIGMETRICS Performance Evaluation Review*, ACM, vol. 42, 2014, pp. 599–600.

[112] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" *IEEE Transactions on Communications*, vol. 64, no. 2, pp. 715–722, 2016.

[113] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*, IEEE, 2012, pp. 2766–2770.

[114] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[115] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3709–3716, 2018.

[116] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Towards collaborative inferencing of deep neural networks on internet of things devices," *IEEE Internet of Things Journal*, 2020.

[117] M. Abadi *et al.*, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015.

[118] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 754–768.

[119] Docker Inc., *Docker: Enterprise application container platform*, https://www.docker.com/, [Online; accessed 4/4/21].

[120] J. Choi, W. J. Jeon, and S.-C. Lee, "Spatio-temporal pyramid matching for sports videos," in *ICMR'8*, ACM, 2008, pp. 291–297.

[121] G. Farnebäck, "Two-frame motion estimation based on polynomial expansion," Springer, 2003, pp. 363–370.

[122] H. Kuehne, H. Jhuang, E. Garrote, T. Poggio, and T. Serre, "Hmdb: A large video database for human motion recognition," in *ICCV'11*, IEEE, 2011, pp. 2556–2563.

[123] NVIDIA, *Nvidia jetson tx*, http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html, [Online; accessed 4/4/21], 2017.

[124] F. Chollet *et al.*, *Keras*, https://github.com/fchollet/keras, 2015.

[125] T. A. S. Foundation, *Apache avro*, https://avro.apache.org, [Online; accessed 4/4/21], 2017.

[126] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Real-time image recognition using collaborative iot devices," in *ReQuEST '18*, ACM, 2018.

[127] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[128] R. Hadidi, J. Cao, T. Kirshna, M. S. Ryoo, and H. Kim, "An edge-centric scalable intelligent framework to collaboratively execute dnn," *SysML Conference Demo, Stanford, CA*, 2019.

[129] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Robustly executing dnns in iot systems using coded distributed computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ACM, 2019, p. 234.

[130] R. Hadidi, B. Asgari, J. Cao, Y. Bae, D. E. Shim, H. Kim, S.-K. Lim, M. S. Ryoo, and H. Kim, *Lcp: A low-communication parallelization method for fast neural network inference in image recognition*, 2020. arXiv: 2003.06464 [eess.SP].

[131] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, *Reducing inference latency with concurrent architectures for image recognition*, 2020. arXiv: 2011.07092 [cs.CV].

[132] V. Vujović and M. Maksimović, "Raspberry pi as a sensor web node for home automation," *Computers & Electrical Engineering*, vol. 44, pp. 153–171, 2015.

[133] R. Grimmett, *Raspberry Pi robotics projects*. Packt Publishing Ltd, 2015.

[134] A. G. Millard, R. Joyce, J. A. Hilder, C. Fleşeriu, L. Newbrook, W. Li, L. J. McDaid, and D. M. Halliday, "The pi-puck extension board: A raspberry pi interface for the e-puck robot platform," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2017, pp. 741–748.

[135] I. Brand, J. Roy, A. Ray, J. Oberlin, and S. Oberlix, "Pidrone: An autonomous educational drone using raspberry pi and python," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, 2018, pp. 1–7.

[136] S. Wilson, R. Gameros, M. Sheely, M. Lin, K. Dover, R. Gevorkyan, M. Haberland, A. Bertozzi, and S. Berman, "Pheeno, a versatile swarm robotic research and education platform," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 884–891, 2016.

[137] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[138] H.-T. Kung, "Why systolic architectures?" *IEEE computer*, vol. 15, no. 1, pp. 37–46, 1982.

[139] JEDEC, *Jedec standard: Low power double data rate 2 (lpddr2)*, https://www.jedec.org/sites/default/files/docs/JESD209-2B.pdf, [Online; accessed 4/4/21], 2019.

[140] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, 2018, pp. 1–8.

[141] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[142] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML'17*, ACM, 2015, pp. 448–456.

[143] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[144] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[145] J. Redmon, *Darknet: Open source neural networks in c*, pjreddie.com/darknet, 2013–2016.

[146] Makerhawk, *Um25c usb power meter*, makerhawk.com, [Online; accessed 4/27/21], 2019.

[147] Xilinx Inc., *Pynq: Python productivity for zynq*, pynq.io, [Online; accessed 4/27/21], 2019.

[148] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2018, pp. 620–629.

[149] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Musical chair: Efficient real-time recognition using collaborative iot devices," *arXiv preprint arXiv:1802.02138*, 2018.

[150] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.

[151] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

[152] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[153] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[154] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," Cornell University, Tech. Rep., 1999.

[155] D. J. Watts, "Networks, dynamics, and the small-world phenomenon," *American Journal of sociology*, vol. 105, no. 2, pp. 493–527, 1999.

[156] M. E. Newman and D. J. Watts, "Renormalization group analysis of the small-world network model," *Physics Letters A*, vol. 263, no. 4-6, pp. 341–346, 1999.

[157] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[158] Wikipedia, *Hypergraph*, https://en.wikipedia.org/wiki/Hypergraph, [Online; accessed 4/11/21], 2019.

[159] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: Applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[160] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

[161] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Proc. of ICVGIP*, 2008.

[162] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *arXiv preprint arXiv:1905.01392*, 2019.

[163] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[164] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017, https://pytorch.org.

[165] S. Li, M. A. Maddah-Ali, Q. Yu, and A. S. Avestimehr, "Coded mapreduce," in *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, 2015, pp. 964–971.

[166] R. Hadidi, J. Cao, and H. Kim, "Creating robust deep neural networks with coded distributed computing for iot systems," *arXiv preprint arXiv:2104.04447*, 2021.

[167] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[168] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.

[169] G. Inc., *Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015*, https://www.gartner.com/newsroom/id/3165317, [Online; accessed 4/4/21], 2015.

[170] F. B. Insights, *Internet of things (iot) market size, share and industry analysis by platform*, https://www.fortunebusinessinsights.com/industry-reports/internet-of-things-iot-market-100307, [Online; accessed 4/4/21], 2019.

[171] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1738–1762, 2019.

[172] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Real-time image recognition using collaborative iot devices," in *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-designing Pareto-efficient Deep Learning*, ACM, 2018, p. 4.

[173] Y. Bae, R. Hadidi, B. Asgari, J. Cao, and H. Kim, "Capella: Customizing perception for edge devices by efficiently allocating fpgas to dnns," in *2019 International Conference on Field-Programmable Logic and Applications (FPL)*, IEEE, 2019.

[174] M. L. Merck, B. Wang, L. Liu, C. Jia, A. Siqueira, Q. Huang, A. Saraha, D. Lim, J. Cao, R. Hadidi, *et al.*, "Characterizing the execution of deep neural networks on collaborative robots and edge devices," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, IEEE, 2019, p. 65.

[175] R. Hadidi, J. Cao, M. L. Merck, A. Siqueira, Q. Huang, A. Saraha, C. Jia, B. Wang, D. Lim, L. Liu, and H. Kim, "Understanding the power consumption of executing deep neural networks on a distributed robot system," *Algorithms and Architectures for Learning in-the-Loop Systems in Autonomous Flight, International Conference on Robotics and Automation (ICRA) 2019*, 2019.

[176] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Collaborative execution of deep neural networks on internet of things devices," *arXiv preprint arXiv:1901.02537*, 2019.

[177] J. Cao, R. Hadidi, J. Arulraj, and H. Kim, "Video analytics from edge to server: Work-in-progress," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis Companion*, ser. CODES/ISSS '19, New York, New York: Association for Computing Machinery, 2019, ISBN: 9781450369237.

[178] P. Corcoran and S. K. Datta, "Mobile-edge computing and the internet of things for consumers: Extending cloud computing and services to the edge of the network," *IEEE Consumer Electronics Magazine*, vol. 5, no. 4, pp. 73–74, 2016.

[179] R. Hadidi, N. S. Ghalehshahi, B. Asgari, and H. Kim, "Context-aware task handling in resource-constrained robots with virtualization," 2021. arXiv: 2104.04563 [cs.RO].

[180] Z. Fu, Y. Mao, D. He, J. Yu, and G. Xie, "Secure multi-uav collaborative task allocation," *IEEE Access*, vol. 7, pp. 35 579–35 587, 2019.

[181] S. Jijina, A. Amyette, N. Shoghi, R. Hadidi, and H. Kim, "Understanding the software and hardware stacks of a general-purpose cognitive drone," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 212–214.

[182] R. Hadidi, S. Jijina, N. Shoghi, B. Asgari, A. Adriana, and H. Kim, "Quantifying the design-space tradeoffs in autonomous drones," in *26thd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2021, pp. 661–673.

[183] T. X. Tran, A. Hajisami, P. Pandey, and D. Pompili, "Collaborative mobile edge computing in 5g networks: New paradigms, scenarios, and challenges," *IEEE Communications Magazine*, vol. 55, no. 4, pp. 54–61, 2017.

[184] L. A. Grieco, A. Rizzo, S. Colucci, S. Sicari, G. Piro, D. Di Paola, and G. Boggia, "Iot-aided robotics applications: Technological implications, target domains and open issues," *Computer Communications*, vol. 54, pp. 32–47, 2014.

[185] M. Erdelj, M. Król, and E. Natalizio, "Wireless sensor networks and multi-uav systems for natural disaster management," *Computer Networks*, vol. 124, pp. 72–86, 2017.

[186] M. Quaritsch, E. Stojanovski, C. Bettstetter, G. Friedrich, H. Hellwagner, B. Rinner, M. Hofbaur, and M. Shah, "Collaborative microdrones: Applications and research challenges," in *Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems*, 2008, pp. 1–7.

[187] N. Michael, S. Shen, K. Mohta, V. Kumar, K. Nagatani, Y. Okada, S. Kiribayashi, K. Otake, K. Yoshida, K. Ohno, *et al.*, "Collaborative mapping of an earthquake damaged building via ground and aerial robots," in *Field and service robotics*, Springer, 2014, pp. 33–47.

[188] A. Bechar and C. Vigneault, "Agricultural robots for field operations: Concepts and components," *Biosystems Engineering*, vol. 149, pp. 94–111, 2016.

[189] H Anil, K. Nikhil, V Chaitra, and B. G. Sharan, "Revolutionizing farming using swarm robotics," in *2015 6th International Conference on Intelligent Systems, Modelling and Simulation*, IEEE, 2015, pp. 141–147.

[190] Y Baudoin and M. K. Habib, *Using robots in hazardous environments: Landmine detection, de-mining and other applications*. Elsevier, 2010.

[191] M. D. de Assuncao, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, pp. 1–17, 2018.

[192] S. Golodetz, T. Cavallari, N. A. Lord, V. A. Prisacariu, D. W. Murray, and P. H. Torr, "Collaborative large-scale dense 3d reconstruction with online inter-agent pose optimisation," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 11, pp. 2895–2905, 2018.

[193] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size*, 2016. arXiv: 1602.07360 [cs.CV].

[194] M. Tan and Q. V. Le, *Efficientnet: Rethinking model scaling for convolutional neural networks*, 2020. arXiv: 1905.11946 [cs.LG].

[195] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, *Aggregated residual transformations for deep neural networks*, 2017. arXiv: 1611.05431 [cs.CV].

[196] G. Marcus, "Deep learning: A critical appraisal," *arXiv preprint arXiv:1801.00631*, 2018.

[197] T. Nagel, "What is it like to be a bat?" *The Philosophical Review*, vol. 83, no. 4, pp. 435–450, 1974.